In *Artificial Neural Networks*, Kohonen, et. al. (eds), Elsevier Science Publishers, pp. 729-734, 1991.

AN EFFICIENT STATIC TOPOLOGY FOR MODELING ASOCS

George L. Rudolph and Tony R. Martinez

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

ASOCS (Adaptive Self-Organizing Concurrent Systems) are a class of connectionist computational models which feature self-organized learn-ing and parallel execution. One area of ASOCS research is the development of an efficient implementation of ASOCS using current technology. A result of this research is the Location-Independent ASOCS model (LIA). LIA uses a parallel, asynchronous network of in-dependent nodes, which leads to an efficient physical realization using current technology. This paper reviews the behavior of the LIA model, and shows how a static binary tree topology efficiently supports that behavior. The binary tree topology allows for *O(log(n))* learning and execution times, where *n* is the number of nodes in the network.

1. INTRODUCTION

ASOCS (Adaptive Self-Organizing Concurrent Systems) is a class of connectionist computational models [2],[4]. ASOCS models support efficient computation through self-organized learning and parallel execution. ASOCS has the same overall goals as other ANN (artificial neural network) models [8]. However, the underlying learning and execution mechanisms for ASOCS differ significantly from those of other ANN models. Current ASOCS models are based on networks of digital nodes rather than analog nodes. The networks store and manipulate digital representations of data. ASOCS learning is incremental rather than training-set style. Examples are incrementally presented and incorporated into an adaptive logic network in a parallel and self-organizing manner, with priority given to the most recent example. The system resolves inconsistencies and generalizes as each example is presented. ASOCS incorporates topological adaptation into the learning algorithm, rather than operating with an initially fixed topology. In execution mode, an ASOCS network operates as an asynchronous, parallel hardware circuit. The network maps input from the environment to output data based on the previously learned examples.

Two significant advantages of ASOCS are: a) an ASOCS net is guaranteed to learning any arbitrary set of legal examples, and b) learning time is fast and bounded, being linear in the depth of the network and logarithmic in the number of nodes. Target applications for ASOCS include adaptive logic devices, robotics, embedded systems, and pattern recognition (standard ANN applications). A detailed discussion of ASOCS is beyond the scope of this paper, but is available in the literature [3],[4],[5].

One area of research interest is the development of an efficient implementation of ASOCS using current technology. A proof-of-concept ASOCS chip was developed at UCLA [1], however other more efficient implementations are of interest. A result of this research is the ASOCS model called Location-Independent ASOCS (LIA) [6]. LIA uses a parallel, asynchronous network of independent nodes, which leads to an efficient physical realization using current technology.

This paper reviews the behavior of LIA, and shows how a static binary tree topology efficiently supports that behavior. Section 2 discusses how to describe a model in terms of structure and behavior. Section 3 section briefly defines basic LIA mechanisms, execution and learning modes. Section 4 discusses location independence. Section 5 describes the physical topology with examples of operation. Section 6 concludes the paper.

2. Describing the LIA Model: Structure and Behavior

The critical function of a particular structure is to support efficiently the model's behavior. *Structure* as defined here means the *node* (internal) structure and the network *topology*, or interconnect between the computing nodes.

The topology must support the behavior of the nodes and the network as a whole. Topologies are often categorized according to form and behavior. There are many different forms, but two basic behaviors--*static* and *dynamic*. When discussing models and potential implementations, a distinction between logical and physical topologies is a useful one. The *logical topology* refers to the topology used in the model. The *physical topology* refers to the implementation (physical realization) of the model topology.

Described in these terms, previous ASOCS models map a dynamic logical topology onto a dynamic physical topology. LIA uses learning and execution algorithms similar to those of previous models. The benefit of LIA is the efficiency of a static physical topology. LIA maps a dynamic logical topology onto a static physical topology in such a way that the dynamic logical behavior is efficiently preserved.

3. BASIC MECHANISMS AND BEHAVIOR

The underlying mechanisms of LIA are the instance, the instance set, and their network representation. The basic unit of knowledge in LIA is the instance. An *instance* is composed of a conjunction of boolean input variables and an output variable. The output variable specifies what the network output should be for that variable whenever the input variables are matched by the environment. Two examples of instances are the following:

AB' => Z'
AB => Z

The first instance specifies that whenever A is high and B is low, Z should be low. The second instance specifies that whenever A and B are both high, Z should be high. LIA handles instances with multiple output variables. In order to simplify discussion of the model in this paper, examples use only one output variable Z.

An instance *is matched by* (or *matches*) an environment (or state of the environment) if and only if, for all variables in the instance's input list, the value of the variable in the environment is the same as the value specified by the instance. The instance A => Z is matched by the environment ABCD, but is not matched by A'BCD, for example. An instance X *covers* another instance Y if and only if all states of the environment that match Y also match X. The instance AB => Z covers the instance ABC => Z, but does not cover the instance A'B => Z. An instance X *contradicts* another instance Y (and vice versa) if and only if a) X and Y specify opposite network outputs and b) there exists at least one possible state of the environment by which X and Y are simultaneously matched. The instance AB'C => Z contradicts the instance AB' => Z': any state of the environment in which A is high and B is low matches simultaneously both instances.

Previous ASOCS models use a distributed hierarchy of 2-input nodes in their networks, in which each node represents only part of an instance. In an LIA network, each node stores a complete instance. Figure 1 shows a network that has stored the two instances listed above. Nodes are numbered for reference only. The topology (how the nodes interconnect) is not specified here in order to emphasize that the behavior of the model does not constrain the topology. Topology is discussed later.

The collection of all instances currently stored in a network is called the *instance set*. The instance set represents the function currently executed (computed) by the network.

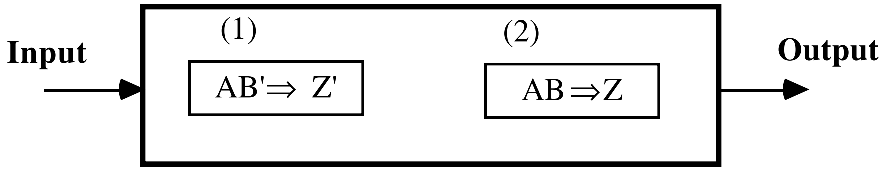

Figure 1. An LIA Network (topology unspecified)

3.1. Execution Mode

In execution mode, the network receives input from the environment and computes an output. The operation of the network is parallel, asynchronous, and the nodes are independent of one another. Each node in the network receives a "broadcast" of the environmental input and checks to see if its input variables are matched. A node whose input variables are matched by the environment activates its output, and that value becomes the value of network. A node whose variables are not matched by the environment does nothing. The following two examples illustrate how LIA operates in execution mode. Assume the network in figure 1 is in execution mode.

The input is **ABCD.** Node 1 compares the environment to its input variables. Node 1 is not matched because B is high in the current environment. Node 1, therefore, does nothing more. Node 2, however, is matched because A and B are high (C and D are *don't care* variables for nodes 1 and 2). Node 2 therefore sends out Z as the value of the network.

It is possible for more than one node to be matched simultaneously by the environmental input. However, the learning algorithm assures that the network is always consistent (without contradictions). Thus, any nodes simultaneously matched by the environment have consistent outputs.

The input is **A'B'C'D'.** Node 1 is not matched because A is currently low. Therefore, node 1 does nothing. Similarly, node 2 is not matched because A (and/or B) is currently low. It does nothing. In this case, none of the nodes in the network matches the input. The network outputs a *don't know*, indicating that it does not know what the output should be. Along with this, several options exist for generalizing on the input in order to come up with a "best guess" answer, such as nearest-match, stochastic selection, etc.

3.2. Learning Mode: Changing The Network Function

The goal of learning is to store a consistent, efficient representation of instances (including generalization) which have been presented [7]. This includes changing the network function. In learning mode, both the input and the desired network output (together called the *new instance*) are broadcast to the nodes. As in execution mode, the operation of the network is parallel, asynchronous, and the nodes are independent. Each node receives a broadcast of the new instance, compares its stored instance with the new instance, and may change its function based on the result. A node may change its instance, may delete itself from the network, or not change at all.

Following are four limited examples of how learning takes place. For an in-depth discussion of LIA learning see [7]. Assume the initial network state in figure 1. The addition of each of the following four instances in order illustrates how a network operates in learning mode:

(1) A'B'C => Z
(2) A' => Z'
(3) A'B' => Z'
(4) AB'CD' => Z

A'B'C => Z. Because their respective A's are opposite from the new instance, nodes 1 and 2 do not cover the new instance, and are not contradicted by it. Thus, neither node makes any changes. However, a new node representing the new instance must be added to the network. The new network is shown in figure 2.

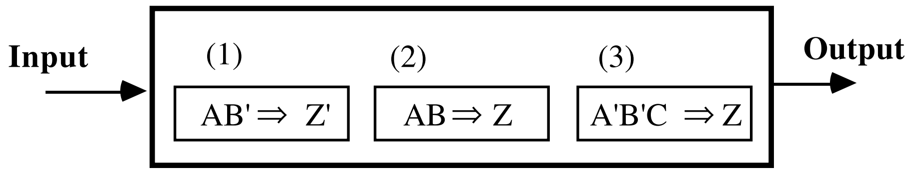

Figure 2. Network After Instance A'B'C => Z

A' => Z'. Nodes 1 and 2 do not cover the new instance, nor are contradicted by it. Node 3, however, is completely contradicted. It specifies that when A and B are low and C is high, Z must be high. The new instance specifies that anytime A is low (regardless of B or C), Z must be low. In this case, node 3 self-deletes from the network, and a new node 3 is added that stores the new instance. The new network is shown in figure 3.

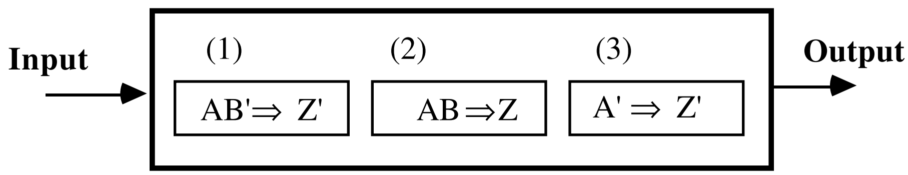

Figure 3. Network After Instance A' => Z'

A'B' => Z'. Nodes 1 and 2 do not cover the new instance, nor are they contradicted by it. Node 3, on the other hand, covers the new instance--the network already sets "Z low" for all states that match the new instance. Therefore, the new instance is not added. The network remains as in figure 3.

AB'CD' => Z. Nodes 2 and 3 do not cover the new instance, nor are they contradicted by it. Node 1, however, is partially contradicted. Node 1 specifies that any time A is high and B is low, Z must be low, regardless of the value of C or D. The new instance specifies that whenever A and C are high, and B and D are low, Z must be high.

The network must be modified to represent correctly the new instance, while preserving the non-contradicted information in node 1. In this case, the instance in node 1 is modified to become two instances: AB'C' => Z' and AB'D => Z'. (These two instances cover the information in the instance in node 1 that is not explicitly contradicted by the new instance.) Node 1 initiates the addition of two nodes to the network, each of which stores one of the instances. Node 1 then deletes itself from the network. The new instance must now be added to the network. Since node 1 is free, node one stores the new instance. Figure 4 shows the final network.

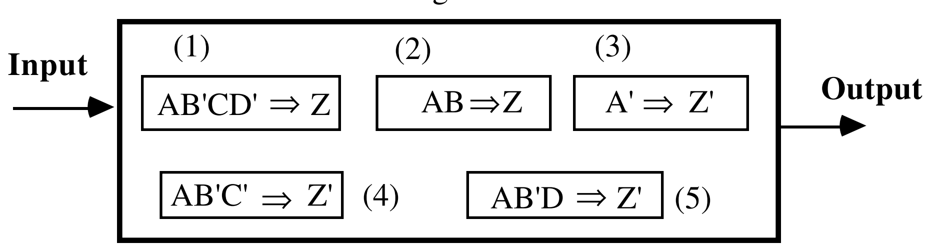

Figure 4. Network After Instance AB'CD' => Z

4. LOCATION INDEPENDENCE

The fact that the operation (in either mode) of each node is independent of any other node is central to the strengths of the LIA model. *Node independence* exists in two ways--functional independence and location independence. *Functional independence* means that the function computed by a particular node is independent of the function computed by any other node. *Location independence* means that the physical position of a node in the network does not determine the node's function nor affect the network's overall function. Functional independence exists because each node stores a complete instance. All of the information about an instance is locally available at a single node. Functional independence leads to location independence.

Location independence in the logical model has important implications for implementation. The physical position of a node (or an instance) does not matter, therefore nodes (instances) can be shuffled easily around the network. No particular logical topology is required by the model, therefore any desired physical topology may be used. *Freed from functional (and locational) constraints, the logical topology may be designed to accommodate the constraints and needs of the physical topology.*

5. PHYSICAL TOPOLOGY

The physical topology must support parallel broadcast and parallel gather operations. A binary tree topology is natural since it a) has fast [$O(log(n)$, where n is the number of nodes in the net] broadcast and gather times, b) has a regular topology that scales well, and c) has a simple, parsimonious form. This section illustrates how learning occurs, and then gives an example of operation in execution mode. A more in-depth discussion occurs in [7].

Learning Example: Figure 5 shows the network from the previous learning example in tree form. Free nodes beyond the frontier of the network exist but are not shown. Figure 5a shows the same initial network as before. Again, the four instances to be added are the following:

(1) A'B'C => Z
(2) A' => Z'
(3) A'B' => Z'
(4) AB'CD' => Z

A'B'C => Z. The new instance is broadcast to the network. The broadcast is received by the root node, which then sends the broadcast on to node 2, and performs its comparison as indicated earlier. Node 2 receives the broadcast and performs its comparison as indicated earlier. Assume that the broadcast terminates at a node when all the node's children are free. As neither node matches the instance, neither node needs to change, and therefore, neither node sends a response.

When no response is received, the new instance is rebroadcast to the network with a flag indicating that it must be added. A single node is uniquely selected (shortest path to free node, etc.) and allocated to store the new instance (figure 5b). When this is complete, the network is ready to learn the next instance.

A' => Z'. When node 3 receives the broadcast, it self-deletes (marks itself as free) because it is completely contradicted. Since none of the nodes cover the new instance, the new instance must be added. Since node 3 is free, it stores the new instance (figure 5c).

A'B' => Z'. When node 3 receives this broadcast, it sends a response back to its parent (the root) indicating that it --or at least one node it its subtrees--covers the new instance. (If at least one node covers a new instance, exactly which one, or how many are immaterial.) None of the other nodes respond, so the root node passes the response it received from node 3 out of the network. The new instance is disregarded, and no changes are made to the network. The network is ready for the next instance (figure 5c).

AB'C'D => Z. Node 1 is partially contradicted by the new instance, and calculates the new instances that need to be added. None of the other nodes cover, nor are contradicted by, the new instance.

Node 1 initiates the addition of two nodes to the network: one node stores AB'C' => Z' and the other node stores AB'D => Z'. Node 1 sends AB'C => Z to its left child, and AB'D' => Z' to its right child (nodes 2 and 3) respectively. (If there were more, the instances would be split into two groups and sent to the children. Node 2 is allocated already, but its left child is free. The left child stores the instance AB'C => Z. Node 3 is allocated already, but its left child is free. Node 3's left child stores AB'D' => Z'. Node 1 then deletes itself from the network, but when the new instance is added, node 1 is allocated to store the new instance (figure 5d).

Two operations, parallel allocation of new nodes and deletion deserve particular mention. First, parallel allocation of new nodes can take place without contention because different subtrees are unique. When a node determines that more nodes (instances) need to be added to the network, all of the information about those instances is calculated by that node. The node then broadcasts that information to the children in its subtrees. If a particular subtree is full, the information can be passed up to the parent and over to another subtree, until sufficient free nodes are found and allocated. Second, when a node deletes from the network it marks itself as "free". The same node (at some later time) may be allocated to store another instance. In particular, "holes" which are created in the tree by deletions may be filled by other instances in such a way that the tree is compacted (made full and balanced). This improves the speed of learning and execution, and makes efficient use of nodes.

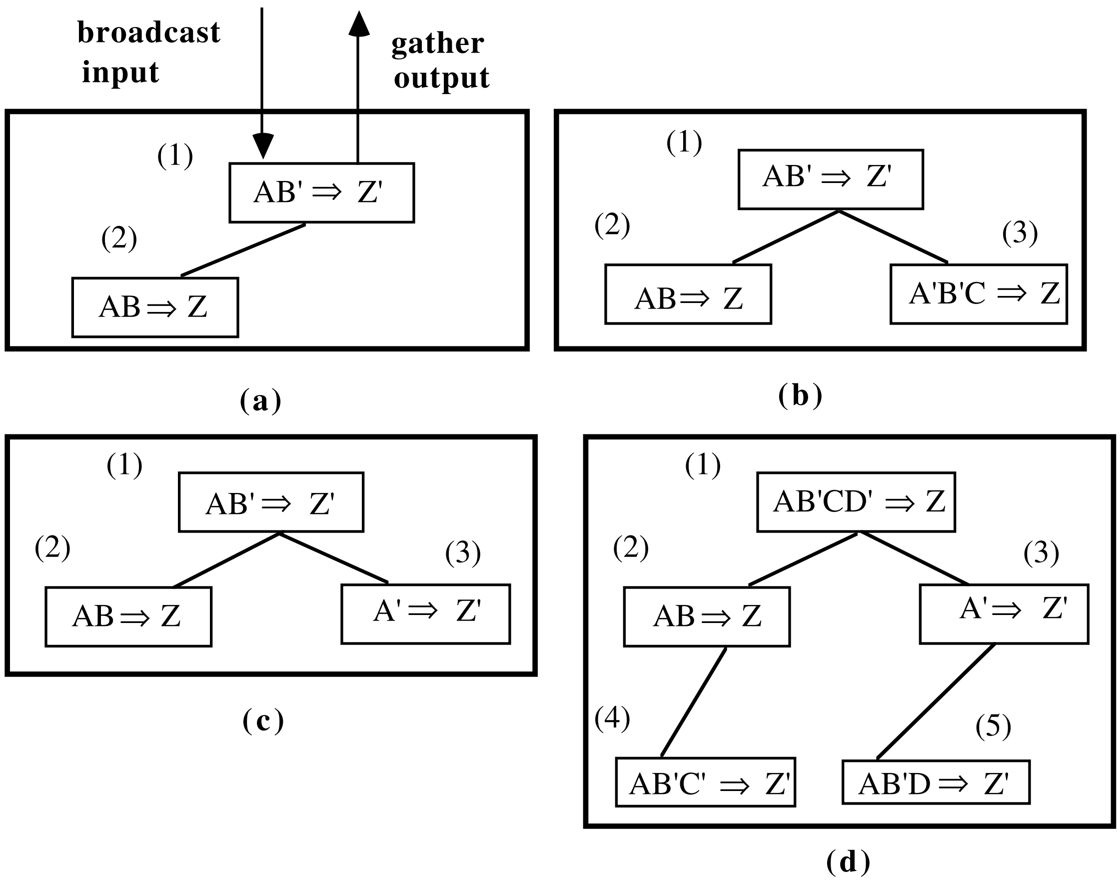

Figure 5. Example LIA Network With Tree Topology

Execution Example: Assume that the network in figure 5d is in execution mode, and the the environmental input is **AB'CD'**. Node 1 receives the broadcast and sends it to nodes 2 and 3. Nodes 2 and 3 send the broadcast on to nodes 4 and 5. Nodes 1, 2, 3 and 4 do not match this input, but node 5 matches. Node 5 sends "Z high" to node 3. Node 3 combines "Z high" with "no response" and sends "Z high" to node 1. Node 4 has no response, so node 2 combines "no response" with "no response" and does nothing further. Node 1 combines "no response" from node 2 with "Z high" from node 3, and sends "Z high" out as the output of the network.

6. CONCLUSION

The LIA model uses a static binary tree topology for a network of independent nodes. Each node stores a complete instance, and instances may be shuffled easily among nodes. This leads to an efficient physical realization using current technology.

LIA uses an m-input logic gate, rather than the 2-input logic gate used in previous models.

Thus, an LIA node is slightly more complex than nodes in previous models. Also, because each node stores a complete instance, LIA's internal representation of instance sets is less distributed than representations in previous ASOCS models [7]. However, the main strength of LIA is that it is efficiently implementable using current technology.

Current research for LIA (and for ASOCS) focusses on using LIA in solving applications, extending the overall applicability of ASOCS, improving generalization abilities, and continuing development of a hardware implementation.

REFERENCES

[1] Chang J., and J.J. Vidal, Inferencing in Hardware. *Proceedings of the MCC University Research Symposium*, Austin, TX, 1987.

[2] Martinez, T.R., *Adaptive Self-organizing Logic Networks*. UCLA Ph.D. Disser-tation, (1986).

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

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

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

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

[7] Rudolph, G., *A Location-Independent ASOCS Model.* BYU Master's Thesis, January 1991.

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