In Proceedings of the *1987 Systems Man and Cybernetics Conference*, pp. 290-296, 1987.

Models of Parallel Adaptive Logic

Tony Martinez
Brigham Young University

Abstract: This paper overviews a proposed architecture for adaptive parallel logic referred to as ASOCS (Adaptive Self-Organizing Concurrent System). The ASOCS approach is based on an adaptive network composed of many simple computing elements which operate in a parallel asynchronous fashion. Problem specification is given to the system by presenting if-then rules in the form of boolean conjunctions. Rules are added incrementally and the system adapts to the changing rule-base. Adaptation and data processing form two separate phases of operation. During processing the system acts as a parallel hardware circuit. The adaptation process is distributed amongst the computing elements and efficiently exploits parallelism. Adaptation is done in a self-organizing fashion and takes place in time linear with the depth of the network. This paper summarizes the overall ASOCS concept and overviews three specific architectures.

Introduction

As deficiencies in sequential "von Neumann" machines become more apparent, work in massively parallel architectures has increased. This paper presents an architectural model for a class of special-purpose parallel processors referred to as ASOCS (Adaptive Self-Organizing Concurrent Systems). Intended applications include adaptive logic devices, process control, robotics, system malfunction management and AI applications of logic reasoning.

ASOCS processing is symbolic and currently supports propositional logic calculations, performed combinatorially in a parallel network of programmable gates. The system is based on digital logic to take advantage of current technology and VLSI advances.

The ASOCS approach is based on an adaptive network composed of many simple computing elements (nodes) which operate in a combinatorial and asynchronous fashion. The nodes operate on boolean inputs producing boolean outputs. Control and processing in the network is distributed amongst the network nodes. Adaptation and data processing form two separate phases of operation. During processing, the network acts as a parallel network of Boolean gates. Logic predicates which have been implicitly "stored" by the connections of the network are processed in data-flow fashion to extract logical inferences.

Declarative problem specification (programming) is obtained by presenting to the system if-then rules expressed as Boolean conjunctions and referred to as instances. New rules are added incrementally to a system rule base. In the present model, when conflicts occur, the rule base is made consistent by giving priority to the newest instance. The rule base need not be stored as in normal applications, but can be implicitly distributed throughout the actual network. Real-world applications using rule based propositional logic have been forthcoming, including a detailed system for in-flight fault maintenance for the Space Shuttle [2].

During adaptation, the network structure and the node functions can change so the overall network function matches the incrementally evolving rule base. As new rules are added to the rule base, the network reconfigures itself to a logic circuit that remains both minimal and consistent with the rule base. Thus, there is no explicit programming of network control. Desired network response is simply presented to the system, following which the network adjusts itself accordingly. The internal configuration is in fact not known by any internal or external agent. An outside view of the model adaptively mapping input to output is shown in figure 1.

ASOCS exploits parallelism at both the processing and adaptation levels. The control of the adaptive process is almost completely distributed amongst the network nodes. Communication takes place between neighboring nodes with only minimal need for centralized processing. Because of its concurrent capabilities the time necessary for network adaptation grows only linearly with the depth of the network.

This paper gives an overview of the abstract ASOCS model. There are a number of potential methods for realizing an ASOCS system. These methods define differing architectures and algorithms which fulfill the basic ASOCS functionality, each having unique features. Three specific ASOCS models (adaptive algorithms) have been formally defined and are informally overviewed in this paper. Detailed and formal presentations of the algorithms are presented in [3] and [4]. Although each model is implemented in a different way, the fundamental goal of adaptive parallel logic inference is accomplished by all three. These three are by no means meant to represent an exhaustive class of ASOCS architectures.

The basic operational sequence of an ASOCS model, which differs somewhat for each implementation, is as follows. A new instance is entered to the system which is currently correctly executing the rule base. The new instance is then broadcast to the logic network. The instance may cause the rule base to become inconsistent, in which case the rule base is made consistent and optionally minimal. Nodes within the network are able to locally determine their ability to aid in the discrimination of the new instance. Nodes are both selected and combined in a distributed manner, resulting in a consistent network. Finally, nodes which are no longer necessary for correct functioning of the network self-delete, thus allowing the network to remain relatively minimal.

ASOCS research was inspired by a revisitation of the seminal work of Rosenblatt [5] on perceptron systems. Verstraete [6] extended perceptron-like techniques into boolean logic devices, in contrast to the threshold unit. His and other [7] research sought centralized methods of programming boolean functions into passive fixed networks. The ASOCS notion seeks to support adaptive networks whose function can change over time, and where control of adaptation is distributed amongst active network nodes in a self-organizing fashion. VLSI implementation and subsequent testing of ASOCS chips is ongoing [1].

This paper first discusses the method of knowledge input, followed by a presentation of the basic architecture. The three adaptive algorithms are then overviewed and compared.

Knowledge Input

The atomic knowledge element of the system is the instance, which is a boolean rule defining what the system should output when confronted with a given input. Each instance is an if-then statement where the antecedent is a conjunction of boolean variables, and the consequent is a single boolean variable. Thus, the instance is a propositional production rule. Instances are universal in that any boolean function can be represented by a set of instances. Following are examples of instances.

$\overline{X}1\ \overline{X}2 \Rightarrow Z1$

$X1\ \overline{X}2\ X3 \Rightarrow \overline{Z}2$

$\overline{X}2\ \overline{X}3\ Z2 \Rightarrow \overline{Z}1$

The $\Rightarrow$ is the implication operator that states if the antecedent is true, then the consequent must become true. (The bar represents the complement of a variable). Note that output variables can be used in instance antecedents, in which case each instance can be viewed as a state transition in a finite state machine. Output variables used in the antecedents are known as feedback variables, and in the current model they can be treated like any other input variable. Though sufficient to support any number of output variables, a single output is considered in this presentation for sake of simplicity.

Whenever the antecedent of an instance is not matched by the input environment, the instance says nothing about what the output of the system should be. Instances are incomplete by nature. The instance $A \Rightarrow Z$ states that Z should be 1 when A is 1 regardless of all other variables. Variables not appearing in an instance are don't care variables for that instance.

In the ASOCS model, instances are input to the system incrementally and the current collection of instances is called the instance set. The instance set is the rule base of the system and adaptation in ASOCS occurs by maintaining the instance set consistent and minimal. By consistent it is meant that no instance in the instance set can contradict another one; i.e. no two instances can specify an opposite output for the same input. Newer instances are given precedence and the contradicted portion of an old instance is deleted from the set. Minimality means that the instance set is stored using a minimal representation. A minimal instance set representation however, is not crucial to the correct functioning of the system, and is not upheld in all models.

An input variable occurring in one instance, and occurring in its complemented form in another instance, is a discriminant variable with respect to the two instances. An instance implying a non-negated variable is a positive instance, while an instance implying a negated variable is a negative instance. This is known as the polarity of an instance. Two instances with the same polarity are concordant. They are discordant instances if they have opposite polarities.

We now consider the relationship between the instance set and a logic network for a single output Z. A logic network fulfills the instance set if, when any positive instance is matched the network output Z is 1, and when any negative instance is matched Z is 0. The next section shows that a positive and negative instance cannot be matched simultaneously in a consistent instance set. If the environment does not match any instance in the set, the network can arbitrarily output a positive or a negated value.

Consistency: A consistent set of instances is one in which no two instances contradict each other. Two instances contradict each other if and only if for some state of the environment, they are both matched, and they are discordant instances. For example, the two instances $A \Rightarrow Z$ and $B \Rightarrow \bar{Z}$ contradict each other when both A and B are 1. Contradictions occur when discordant instances exist without discriminant variables in their antecedents. Two instances with no discriminant variables, can always for some value of the variables both be matched. A corollary to the above is that all instances must contain at least one discriminant variable with respect to each discordant instance.

There are two basic ways by which the instance set is modified due to a contradiction. When the variables of a new instance (NI) are a subset or equal to those of an old instance (OI), the OI is deleted from the IS. For example:

$A\ B\ C \Rightarrow \bar{Z}$ (OI)

$A\ B \Rightarrow Z$ (NI)

By contrast, when there is no discriminant variable between the NI and the OI and the OI is not a subset of the NI, the OI is replaced by one or more modifications of the OI. The addition includes all of the OI which is not contradicted by the NI. Thus, the added instances contain the OI concatenated with one discriminant variable of the NI. For example,

$A\ B \Rightarrow Z$ (OI)

$A\ C\ D \Rightarrow \bar{Z}$ (NI)

will be replaced by

$A\ B\ \bar{C} \Rightarrow Z$

$A\ B\ \bar{D} \Rightarrow Z$

$A\ C\ D \Rightarrow \bar{Z}$

It should be noted that only the NI can cause that a network change be necessary. Instances added to the set in the manner of the last example are already fulfilled by the network since the root instance from which they are constructed was fulfilled.

Minimization: Minimization can only occur between concordant instances. Minimization of boolean sum-of-products forms is well established in the literature and the proofs will not be reproduced here. Minimization is based on repeated application of the following 3 boolean identities:

1. $x \vee xy \equiv x$
2. $xy \vee x\bar{y} \equiv x$

3. $x \vee \bar{x}y \equiv x \vee y$

The resultant minimal set always has the least possible num-ber of instances, and no instance in the set is reducible to an instance with less variables. However, the minimal set is not unique, and there may be many equivalent sets having the same number of instances with differing variable combinations.

Thus, we define a set of instances as minimal if no two concordant instances can be equivalently represented by one instance, or by two instances with fewer variables.

Minimality is not essential to correct functioning of the ASOCS. It can provide savings in space and hardware, but only at the cost of increased adaptation time. The ASOCS models using complete minimization cannot guarantee adaptation in linear time and its use is an optional decision for the implementor.

ASOCS Architecture

The Adaptive Self-Organizing Concurrent System (ASOCS) is made up of boolean inputs and outputs and an input channel for instance input. There is no constraint on the number of outputs in relation to the number of inputs, although only 2^{2^n} unique outputs can exist for n inputs.

Figure 2 shows the internal structure of ASOCS. The two main components are the adaption unit (AU) and the logic network.

During execution mode, only the logic network is active. The input data flows asynchronously through the network with only propagation delays. During adaptation mode, the other parts of the system become active. The AU and the logic network are connected by a broadcast bus. (The use of the word bus does not mean to imply any specific implementation. This and other names used in this paper relate only to functionality and each could be implemented in a variety of ways.) The AU can broadcast to the entire logic network by placing a message on the broadcast bus, but it cannot address a specific node in the network. A node within the network can also place a message on the broadcast bus, which can be read in turn by the AU or any other node.

The AU can feed test data into the network on the presentation path and through the input router, which is controlled by the AU. During adaptation, the input router can get its input from the AU. During execution, the input router passes boolean inputs from the environment to the network. The AU can also monitor the outputs of the network through the test path.

It is also necessary to bind variables, in a flexible manner, to input and output lines. This is done for input and output variables by the input binder and output binder. Thus when new (not previously used) input or output variables are used in a new instance, an input or output line will be allocated for the new variable. The feedback path allows output variables bound in the output binder to be fed back to the input binder.

The adaption unit (AU) guides the logic network through adaptation using global commands, which are messages placed on the broadcast bus. This allows all nodes within the network to work cooperatively by simultaneously executing procedures triggered by the command. Instances are input to the adaption unit.

The global commands used are mentioned in the discussion of the individual algorithms.

Network Node: A single node within the logic network is represented in figure 3. Each node in the network has two basic parts: The control unit and a dyadic dynamic programmable logic module (DPLM). The DPLM is a logic gate which can be dynamically set to a subset of the 16 boolean functions of 2 inputs. During execution mode the DPLM is the only active element.

During adaptation the control unit can change the function of the DPLM. The control unit is a small finite state machine, with some random access memory, and the ability to communicate with the nodes to which it is directly connected. Each network node is identical, except for the contents of its memory, and the current function of its DPLM.

The control unit can send local commands on a bidirectional control path, which connects it to its immediate neighbors. The two nodes which give input to a node are known as the child nodes. The nodes to which a node sends its output are parent nodes. There may be more than one parent node for a given node, but there cannot be more than two children. The two nodes which input to a parent node are called sibling nodes with respect to each other.

ASOCS systems are amenable to fault tolerance by allowing testing and recovery mechanisms to be undertaken between the different nodes of the network. Testing and other activities can be continuously ongoing in the control units of the nodes, since the control units are not used during the normal processing activity of the network. These and other implementation issues are beyond the scope of this paper.

Adaptive Algorithms

The next three sections give brief descriptions of three formally defined algorithms. These are called adaptive algorithms (AA) since they deal with how the system adapts to the input of new instances. The algorithms are labeled as AA1, AA2, and AA3. Each section first discusses any architectural variances from the basic architecture discussed in the last section. This is followed by an informal description of the working algorithm. The descriptions are regrettably short due to the scope of the paper, but are intended to give a flavor of the type of methods used for the decentralized reconfigurations. Although suited to any number of inputs, for sake of simplicity, we continue to assume that there is only one output variable.

Adaptive Algorithm 1: The main variant between the basic ASOCS architecture and that of adaptive algorithm 1 (AA1) is that 3-state DPLM's are used. The DPLM can be set to any of the 8 variations of the AND and OR functions with different inversions of the inputs.

In AA1, the instance set is maintained consistent and minimal in the adaption unit and is stored in a data structure called the Instance Table (IT) (figure 4).

P	N	D
A $\bar{B}$ D	$\bar{A}$ $\bar{B}$ $\bar{E}$	
B C D	B $\bar{C}$ $\bar{D}$ F	

$\bar{A}$ E $\bar{F}$	A B $\bar{D}$	
-	A B $\bar{C}$	
A $\bar{B}$ $\bar{C}$	B $\bar{C}$ $\bar{D}$ $\bar{E}$ F	
A $\bar{B}$ C	B $\bar{C}$ $\bar{E}$ G	
A $\bar{B}$ D	-	

Positive instances are stored in the P column, negative instances in the N column, and the purpose of the D column is explained below. Cells which are empty are marked "-". The order of the stored instances is inconsequential.

When a new instance is input to the AU, it is stored in the first empty location in the instance table. For example, if the NI is a positive instance, then it is stored in the first empty cell in the positive column.

Each node control unit in the network contains a node table (NT). The function of the NT is to store the value of its DPLM output (1, 0, or *?*) for each instance. For example, if a given node outputs 0 for all environment states matching an instance, a 0 is placed in the cell corresponding to that instance. The structure of the NT (i.e., its indexing), in terms of what each cell represents, is the same as that of the instance table. Thus, the third positive cell of every NT, corresponds to the instance in the third positive cell of the instance table. Consequently, a cell in the NT can be in one of four states represented by "1", "0", "*?*", and "-".

P	N	D
1	0	
?	0	
0	?	
-	1	
?	0	
0	1	
0	-	

Figure 5 - Node Table

Figure 5 shows an example of a possible NT which corresponds to the instance table shown in figure 4. When a new instance (NI) is input to the AU, it is presented to the network. A global presentation command is sent to the network giving the polarity of the NI. Presentation is done by setting the data inputs entering the network to the value of the NI. All inputs not corresponding to a variable found in the NI are set to "?". The values flow through the DPLM's and each control unit stores the value which its DPLM outputs in the first empty cell of the node table column corresponding to the NI. Since 3-state values are enter into the network, a node who's DPLM outputs a 1 (or 0) will output 1 (or 0) for all states of the environment that match the NI. If the top node of the network outputs the polarity of the NI, then the network already fulfills the NI and no changes need be made.

However, if the top node output does not match the NI polarity, then the network must be reconfigured to fulfill the instance set. This is done by selecting nodes in the network which can discriminate the NI from all old instances of the opposite polarity. Assume for a given node that the cell corresponding to the NI contains a 0. This node would then discriminate the NI from all discordant instances whose cell contains a 1. Each node can sum up the number of discordant instances which it discriminates from the NI. This value is known as a discriminant count.

At this point the AU broadcasts the selection global command. This initiates a parallel heap sort, where each node sends its discriminant count to its parent, and the parent forwards the maximum count to its parent. Thus, the top node will have received the maximum discriminant count in the entire network in log(n) time where n is the number of network nodes. A downward wave is then initiated, resulting in the selection of the single "best" node in the network. Ties are decided arbitrarily. The selected node sets itself as a growth node. Since a single node is selected, this node can place information on the broadcast bus without bus conflict, which can in turn be read by all other nodes in the network and by the AU. In this way the selected node can convey to the rest of the network the amount of discrimination it does and what still remains to be done. The message placed on the bus is a bit vector containing a 1 for each discordant instance that the node discriminated. Before the beginning of the selection wave, each node sets each cell in its D column to 0. After the bit vector is broadcast, each cell then replaces the contents of its D column with the logical OR of its D column and the broadcast vector. If the new vector still contains 0's a new selection wave is initiated. When each node sums its discriminant count, it only counts instances with a 0 in D cell of the same row, signifying that the instance has not yet been discriminated. The AU updates the D column of the instance table in the same fashion, and is thus able to correctly initiate selection waves. When the D column of the AU contains all 1's then sufficient nodes have been selected to discriminate the NI.

If the maximum discriminant count in the network is 0, or below some threshold, and the D column still contains 0's, then new nodes must be added to the network in order to discriminate the NI. The AU can guide this new node addition since it knows which instances remain un-discriminated. A minimal set of variables necessary to discriminate the NI can be calculated from the instance table and they become inputs to newly allocated nodes, which are also set as growth nodes.

Once sufficient growth nodes are selected, the combination phase ensues. Growth nodes combine, allocating a new parent to which their outputs enter. The DPLM function and the node table of the new parent can be set by the two children nodes without outside intervention. The new parent is then set as a growth node. Combinations can take place concurrently and independently throughout the network. After all growth nodes have combined, a new node which discriminates the NI from all discordant instances will have been created. This node can then be combined, again using only information of the children, with old top node, which correctly discriminates all instances except the NI, to create a new top node which fulfills the total instance set.

After network modification, the final phase of processing is self-deletion. A node, by viewing the contents of its node table, can deduce to what extent it aids in the overall functioning of the network. If a node discovers that it is no longer necessary, for example if it does not discriminate at least one positive instance from a negative instance, then it will self-delete. Upon self-deletion, the node sends local commands to its children for them to also self-delete. Thus large portions of the network can be pruned away, leaving behind a correct network which is at all times relatively close to an optimal network. If a node discovers that it fulfills the entire instance set (it has all 1's in its P column and all 0's in its N column) then all nodes above it can be deleted and it will become the new top node. There are a number of other ways by which self-deletion can take place.

To summarize, in AA1 there is always a single top node for each output variable which is a complete-discriminant node for that variable. When a new instance is added to the set, and if the top node does not already fulfill the new instance, a new node is created which discriminates the new instance from all old discordant instances. This node is created by combining nodes within the network which already discriminate a subset of the old discordant instances from the new instance. If the union of these selected nodes is not sufficient to discriminate the new instance from all discordant instances, then new nodes are allocated which discriminate the remaining instances. Once the new node (which discriminates the new instance from all discordant instances) is built, it can be combined with the old top node (which discriminates all old instances, but not the new instance) creating a new top node which is a complete discriminant node for the given output variable. After the network has reconfigured to fulfill the NI, which takes time linear with the depth of the network, the whole process can iterate and more instances be input to the AU. A detailed presentation of AA1 is given in [4].

<u>Adaptive Algorithm 2</u>: AA2 uses a method much different than that of AA1. Instead of having a single top node for an output variable, there can be a number of top nodes for an output. These are divided into positive and negative nodes. If any positive node outputs a 1, then the output of the network must be a 1. If any negative node outputs a 1, then the output of the network must be a 0. Outputs from top nodes are OR'ed together. In AA2 there is one top node matching each instance in the instance set. Thus the network is maintained as a consistent and minimal instance set. There is no need to maintain the instance set in the AU, since the network implicitly maintains it.

There are two major architectural differences between AA1 and AA2. First, the DPLM in AA2 is a 2-state gate which is always set to the AND function and need not be programmable. Rather than store a table of outputs for all instances, each node stores a list of the variables it conjuncts. A node whose variable list matches an instance is a D-node (discriminant node). The variable list of a D-node is built up by conjunctions of P-nodes (primitive nodes). Each D-node has a polarity which is that of the instance it matches.

The second major difference in AA2 is an added construct called the or-plane. Figure 6 shows a general representation of the structure of the logic network.

Inputs to D-nodes must be from either a P-node, or a literal variable (i.e. A, B, etc.). The D-nodes are the top nodes of the network. However, D-nodes can be found at any depth of the network. The inputs to the or-planes must be from D-nodes.

The outputs of all positive D-nodes enter into the positive or-plane. The outputs of all negative D-nodes enter into the negative or-plane. The or-planes need not be adaptive, The logical structure of the or-planes, which includes the positive and the negative planes, is shown in figure 7.

A 3-node structure handles the or-plane outputs. The middle node is the output node. It outputs 1 if the positive plane outputs a 1, and it outputs 0 if the negative plane outputs a 1.

If both planes output a 0, then the don't know output is active. With this output, one can always know whether an instance is matched or whether the environment state is a don't know state. If both planes output a 1, then there is an error in the network.

As can be seen by viewing the AA2 structure, the algorithm implements a sum-of-products expression. The products are the instances represented by the D-nodes in the network and each D-node is equivalent to one instance in the instance set. The products represented by the D-nodes are summed by the or-planes.

We now discuss the operational sequence of the algorithm. When a new instance (NI) is input to the AU, there is no storage or processing necessary in the AU, since the instance is both stored and maintained implicitly in the network. Presentation is not used to introduce instances to the network as in AA1, thus there is no need for 3-state DPLM's. A NI is simply broadcast to the network as a list of variables (with some variables possibly negated), along with the polarity of the NI. Each node in the network can then directly compare the NI with their own variable list, and independently deduce a necessary action. There is no need for selection waves since nodes can select themselves immediately and independently after viewing the NI.

Nodes execute unique procedures depending on how they match with the NI. For example, if a D-node exactly matches the variables of the NI, but is of the opposite polarity, it simply changes its polarity and redirects its output to the opposite or-plane. This is equivalent to deleting the old instance and adding the NI to the set. If a node's variable list was a superset to the NI, it would self-delete regardless of its polarity. If no node matches the NI, then a new D-node is created to match the NI by combining P-nodes which are a subset of the NI. There are 15 ways in which a node can match a NI, and 8 basic actions nodes can take depending on how they match [3].

<u>Adaptive Algorithm 3</u>: Most of the constructs defined in the previous section, carry over to AA3 and are not reviewed. AA3 is similar, although simpler than AA2. Rather than have one D-node for each instance in the instance set, it is only necessary to have sufficient D-nodes to properly discriminate the instance set. When a new instance is broadcast, only discordant D-nodes compare themselves with the new instance, thus the only test is for contradiction. The main advantage to AA3, besides simplicity, is that fixed interconnections between nodes are possible, allowing for an easier physical implementation.

The DPLM function of each node is always the AND function. There are positive D-nodes, negative D-nodes, and P-nodes. All nodes have one variable child and one node child. A variable child is a literal variable, possibly inverted, which inputs to the node's DPLM. The node child is a network node. Each P-node has two parent nodes. A D-node has no parent nodes. All nodes also have a variable-list and a polarity-flag. Outputs from D-nodes exit the network to the sides and join the or-planes. Since inputs enter at all levels of the network the variable inputs also enter the networks from the side. Figure 8 gives a representation of the AA3 network structure.

The overall structure is that of a binary decision tree (BDT). The bottom node of the network is called the root node and it is unique from all other nodes in the network. It has no inputs, and its DPLM always outputs a "1". Otherwise it is identical to any other node. Figure 9 gives an illustration of a possible internal network structure of AA3. The connections from the D-nodes into the or-plane and from the input binder to the nodes are not shown.

The flexible portion of the network is at the input and output binders. The perpendicular traffic (variable inputs and outputs) needs to be dynamically modifiable. It is by changing the variable inputs to the nodes, and the polarity of the nodes, that the network adapts.

As in the previous algorithm, there is no need to store the instance set in the AU, since it is implicitly stored in the network. However, in AA3 it is impossible to extract the exact instance set, since there is not a unique node for each instance. A NI is broadcast to the network and the nodes initiate a specific action depending on how their variable list and polarity match with the NI. In AA3 only discordant D-nodes can match a NI, and there are only two types of possible action.

If the node's variable list is a superset or equal to the NI, the node simply complements it's polarity flag and redirects its output to the opposite or-plane. If the node's variable list is not a superset of the NI and it is also not discriminated (i.e. there does not exist at least one discriminant variable between the NI and the variable list), then the operation of discriminant variable addition takes place. Briefly, this is accomplished by growing new nodes starting from the matched node using variables from the NI which do not occur in the node's variable list. The matched node becomes a P-node and the parents become D-nodes. As in AA2, all node actions take place concurrently and independently, allowing for high adaptation speed.

Self-deletion follows the network modification, and is defined in one simple operation. Any two sibling D-nodes which have the same polarity both self-delete and their node child becomes a D-node with the same polarity. This operation can continue recursively and in parallel on all D-nodes.

Comparison of Algorithms

In this section we compare the features of the three algorithms discussed above. Software simulation of all three algorithms was used to understand the efficiency of the algorithms for large instance sets. All metrics are dependent on the specific instance set and thus a large number of random instance sets were applied to each algorithm to obtain the average behavior of the different systems. One measure of the total size of the network is number of nodes relative to the number of instances in the current instance set. The number of nodes in an AA1 network is typically equal to the number of instances in the set. AA2 and AA3 usually have two times as many nodes as rules. However, the network depth in AA1 is larger than in the others. The maximum network depth, which is the main bound on speed in an ASOCS system, is equal to the number of bound input variables for both AA2 and AA3.

The main disadvantage of AA1 is the necessity to have memory at each node proportional to the size of the instance set (2 bits per instance). This memory must also grow in proportion to the number of output variables handled by the system. This problem is mitigated in both AA2 and AA3 by use of constant sized variable lists which can be no larger than the number of input variables in the system. Also there is only negligible increase of memory per node for systems with multiple output and feedback variables. AA2 and AA3 have the advantage of using a fixed 2-state gate, and they do not require presentation and selection waves.

In AA1 it is also necessary to both store and maintain the instance set in the adaption unit. In AA2 and AA3 the instance set is both stored and maintained implicitly within the network structure.

AA2 has the unique ability to signal whether the current environment input matches any instance in the set, a more powerful notion than fulfilling the set as previously defined. AA3 loses this ability by using a simpler scheme of only testing for contradiction, and not for minimization. AA3, on the other hand, can be implemented on a network without flexible connections between nodes.

Conclusion

In this paper we have attempted to summarize in a brief fashion the ASOCS computing methodology. ASOCS is an architecture for high speed logical inference with the capability to adapt in a concurrent and distributed fashion. Three specific algorithms were summarized with the intention of giving a flavor for the types of methods available for this type of computing. Current research efforts include categorization of applications and hardware prototyping of the ASOCS architecture.

Bibliography

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

2. Helly, J.J, W.V. Bates, M. Culter, and S. Kelem, "A Represen-tational Basis for the Development of a Distributed Expert Sys-tem for Space Shuttle Flight Control," NASA Technical Memo-randum 58258, (May 1984).

3. Martinez, T. R., "Adaptive Self-Organizing Logic Networks," Ph.D. Dissertation, Computer Science Department, University of California, Los Angeles, CA (May 1986).

4. Martinez, T. R. and J. J. Vidal, "Adaptive Parallel Logic Net-works," To appear in the Journal of Parallel and Distributed Com-puting.

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

6. Verstraete, R.A., "Assignment of Functional Responsibility in Perceptrons," Ph.D. Dissertation, Computer Science Depart-ment, University of California, Los Angeles, CA, (June 1986).

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