Report Number: WUCS-88-26

1989-05-02

Correct Parallel Status Assignments for the Reason Maintenance System

Rosanne M. Fulcomer and William E. Ball

This paper represents a beginning development of a parallel truth maintenance system to interact with a parallel inference engine. We present a solution which performs status assignments in parallel to belief nodes in the Reason Maintenance System (RMS) presented by [3],[4]. We examine a previously described algorithms by [7] which fails to correctly detect termination of the status assignments. Under Petrie's algorithm, termination may go undetected an in certain circumstances (namely the existence of an unsatisfiable circularity) a false detection may occur. We present an algorithm that corrects these problems.
... Read complete abstract on page 2.

Correct Parallel Status Assignments for the Reason Maintenance System

Rosanne M. Fulcomer and William E. Ball

Complete Abstract:

This paper represents a beginning development of a parallel truth maintenance system to interact with a parallel inference engine. We present a solution which performs status assignments in parallel to belief nodes in the Reason Maintenance System (RMS) presented by [3],[4]. We examine a previously described algorithms by [7] which fails to correctly detect termination of the status assignments. Under Petrie's algorithm, termination may go undetected an in certain circumstances (namely the existence of an unsatisfiable circularity) a false detection may occur. We present an algorithm that corrects these problems.

CORRECT PARALLEL STATUS ASSIGNMENTS FOR THE REASON MAINTENANCE SYSTEM

Rosanne M. Fulcomer and William E. Ball

WUCS-88-26

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899

A shorter version of this paper appears in the proceedings of the *1989 International Joint Conference on Artificial Intelligence.*

This work was supported by McDonnell Aircraft Company under contract number Z71021.

Correct Parallel Status Assignment for the Reason Maintenance System*

Rosanne M. Fulcomer

William E. Ball

Department of Computer Science
Washington University
One Brooking Drive
St. Louis, MO 63130

rmf@wucs1.wustl.edu
web@wucs1.wustl.edu

Revised May 2, 1989

Abstract

This paper represents a beginning development of a parallel truth maintenance system to interact with a parallel inference engine. We present a solution which performs status assignments in parallel to belief nodes in the Reason Maintenance System (RMS) presented by [3], [4]. We examine a previously described algorithm by [7] which fails to correctly detect termination of the status assignments. Under Petrie's algorithm, termination may go undetected and in certain circumstances (namely the existence of an unsatisfiable circularity) a false detection may occur. We present an algorithm that corrects these problems.

1 Introduction

A major problem facing AI researchers today is that computation time for inferencing and related activities is extremely expensive. This means large knowledge bases will be almost impossible to use in practical applications that require a fast or predictable response time (e.g. real-time systems). However, by designing and using

*Supported by McDonnell Aircraft Company under contract number Z71021.

efficient parallel algorithms, we hope to generate a significant increase in speed over sequential methods.

We will present a solution which performs status assignments in parallel to belief nodes in the Reason Maintenance System (RMS) presented by [3], [4]. [7] describes an incomplete parallel algorithm using the technique of diffusing computation given by [2].

2 Background

In this section we present a brief overview of both termination detection for diffusing computation problem as presented by [2] and Doyle's RMS. Then, we give Petrie's parallel solution for giving status assignments to belief nodes and show how this solution can lead to incorrect termination detection.

2.1 Diffusing Computation

In the diffusing computation problem, we are given a system consisting of a number of nodes (processors) able to communicate over links. We assume the existence of a node without incoming links and call this node the environment or root. All other nodes are called internal nodes. Initially each node is in a neutral state. A diffusing computation begins when the environment sends a message to one or more successor nodes. This message or set of messages is to be sent only once.

When a neutral internal node receives a message, it becomes *engaged* and may send messages to its successor nodes. At some later time it returns to the neutral state; nodes in the neutral state may not send messages. A node may change from neutral to engaged and back several times during the computation.

A diffusing computation is defined as having terminated when all nodes have reached the neutral state. The computation is such that only a finite number of messages is sent from any internal node. With this restriction, it can be shown that the computation will eventually terminate.

We require that the environment be able to tell that the diffusing computation has terminated. To detect termination, a signaling system is superimposed on the diffusing computation such that the environment will be signalled when the computation is completed. The signaling is restricted to require that from the moment the computation begins to the time the environment is signaled, each link will have carried as many messages in one direction as it has carried signals in the other direction.

The signalling system obeys the following rules:

Rule 1 *When a neutral node receives a message and becomes engaged, it "remembers" the identity of its engager.*

Rule 2 *When an engaged node becomes neutral, it sends a signal to its engager.*

Rule 3 *An engaged node may not become neutral until it has sent a signal for each message it has received; the signal for the message which caused it to become engaged is sent last (as implied by the Rule 2).*

Rule 4 *An engaged node may not become neutral until it has received a signal for each message it has sent.*

From these rules, the following theorems are concluded (see [2]):

Theorem 1 *At all times, a directed path exists from the environment to each engaged node along the edges from engagers to engaged nodes.*

Theorem 2 *A bounded number of steps after the diffusing computation has terminated, the environment will have returned to the neutral state.*

Theorem 3 *When the environment has returned to the neutral state, the diffusing computation has terminated.*

2.2 Review of Doyle's RMS

An RMS is used along with an inference engine (IE) to maintain a consistent set of beliefs and inferences. The inferences are then passed to the RMS, which creates a node for each belief and maintains the dependencies between these beliefs.

Each node in the RMS is to be assigned a status of IN or OUT, where an IN label means the node is believed to be true, and an OUT label means either the truth value of the node is not known or the node is not believed to be true. Associated with each node is a set of justifications, in which each justification contains an INSET and an OUTSET. An INSET contains those nodes which must be believed in order for the node to be labeled IN. An OUTSET contains those nodes which must not be believed in order for the given node to be labeled IN. A justification is valid if every node in its INSET is labeled IN and every node in its OUTSET is labeled OUT. If at least one justification in a node's justification set is valid, the node is labeled IN; otherwise the node is labeled OUT.

The consequences of a node C is the set of all nodes which mention C in one of their justifications. The supporters of the node (supporter-nodes) is the set of nodes which the RMS used to determine the status of the node. For IN nodes, the supporter-nodes are the INLIST $\bigcup$ OUTLIST of its supporting justification. The RMS picks one node from each justification in the justification set to form the supporter-nodes of an OUT node. These nodes are either an OUT node from an INLIST or an IN node from an OUTLIST in a justification.

The ancestors of a node are formed by taking the transitive closure of the supporter-nodes of that node. The set may include the node itself.

2.3 Petrie's Use of Diffusing Computation

To take advantage of the proofs supplied [2] for diffusing computations, [7] proposed considering each node in a RMS to be a separate processor. Justifications are then represented as directed arcs from antecedents to the consequence; only a single consequence is computed by each rule in this representation. **In all figures, a justification is represented by a circle with incoming arcs from antecedents and outgoing arcs to the consequence. A plus next to the antecedent arc indicates an antecedent in the INSET and a minus indicates an antecedent in the OUTSET.** Diffusing computation is used to give a status assignment of IN or OUT to each node corresponding to the labeling that is performed in Doyle's algorithm.

Petrie admits that his computation is incomplete in the same sense that Doyle's is incomplete, which is with respect to unsatisfiable circularities being introduced into the network by a new justification. This in turn creates a graph for which no consistent status assignment can be found.

Each processor stores the set of justifications for that node. Messages are sent to consequences and signals are sent to antecedents in the justification set. When all belief nodes are in a neutral state, one node becomes the environment or root of the diffusing computation when its status changes from OUT to IN by the entrance of a new justification (passed from the IE).

The algorithm begins by the root node issuing a "NIL sweep", which sets the status of its transitive closure of consequences to NIL. This is done using a diffusing computation. Once the "NIL sweep" has terminated, a second diffusing computation is begun for IN/OUT status assignments.

The use of the engager is as discussed in the previous section. Once a node receives its first message from its engager, it checks to see if its status will change. A NIL status will change to IN or OUT, so all statuses will change at least once. If the status is unchanged, a signal is sent back to the predecessor and no messages are sent to the consequences. If the status of the node is changed, it sends messages to its consequences. A processor also replies to any sender if it has already received another message to which it has not replied, i.e. it is engaged and the new message does not change the node's current status assignment. A node replies to the engager (under normal circumstances) when it receives replies from all of its consequences or successors.

To detect unsatisfiable circularities, Petrie proposes the following solution. Along with a status message, an antecedent node sends a list of ancestors to its consequences. This list will include the antecedent when it is sent. If the consequent node appears in an ancestor list it receives, within a message that changes its status, it signals a "trouble reply" to its engager. Thus a node can determine if its current status depends on itself. Petrie states that if an engaged node switches status twice as a result of being its own ancestor then it is involved in an unsatisfiable circularity. We will also use this observation in our algorithm.

2.3.1 Problems with Petrie's Algorithm

In this section, we point out three problems with Petrie's solution.

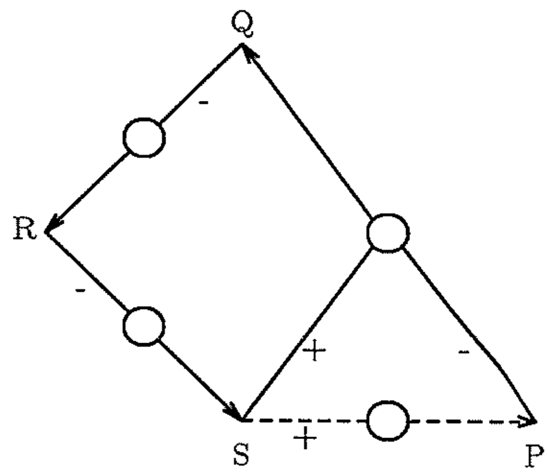

Figure 1.

Problem 1 *Incorrect definition of environment node.*

Petrie does not define the environment node such that it has no incoming edges, which permits termination to be detected. In his computation, the environment node may have incoming edges, as in Figure 1. Since S is both a supporter of P *and* in the transitive closure of consequences of P, P may receive incoming messages.

Problem 2 *Possible false termination detection in the presence of unsatisfiable circularities.*

In detecting unsatisfiable circularities, Petrie's solution allows the consequence to signal a "trouble reply" to its engager if the consequence appears in an ancestor list within a message which changes its status. But if the consequence signals "trouble" to its engager, it can violate all the rules regarding signalling the engager. Such a signal must be sent only when the consequence becomes neutral; only after the consequence has sent signals for all other messages it has received; and only after it has received signals for all messages it has sent.

Even if the sending of the trouble signal is otherwise legal, the sending of this signal makes the consequence neutral. Since it must change status, it will have to send messages to transmit its new status to its consequences. These messages violate the requirement that neutral nodes send no messages.

The consequences which were engaged by the troubled node above may still be computing their own statuses. The signaling of the trouble reply has cut off the directed path from the environment to such consequences. The environment may detect termination while computation is still going on. Thus it is possible to falsely detect termination.

Problem 3 *Non-termination of the underlying computation.*

The termination-detection algorithm for diffusing computations requires that all nodes send a finite number of messages. In Petrie's algorithm, if an unsatisfiable circularity occurs all nodes on the cycle will compute forever, changing their statuses and sending update messages to their consequences on the cycle. Petrie's algorithm detects the existence of such cycles but does not stop the status-assignment processing of the nodes on the cycle.

3 Improved Algorithm

The solution we present also uses diffusing computation to perform the status assignments. We use the same mapping Petrie proposed, i.e. each node in the RMS is a separate processor and justifications are represented as directed arcs from antecedents to a single consequence. We will use the term *cycle* to mean the circular passing of messages. A better term would be circularity, but its use becomes too cumbersome. For our purposes, a cycle may become an unsatisfiable circularity or may be satisfied.

In our solution, the environment or root node does **not** represent a belief. It is a separate node which is responsible for receiving the justification from the IE and passing the first message to the consequece of that justification to begin the computation. All belief nodes will be considered internal nodes and will compute the same algorithm. The node whose status changes because of the incoming justification will be called the *head node*. In this section we will discuss the actions of an internal node in general, including the handling of unsatisfiable circularities. Then we will

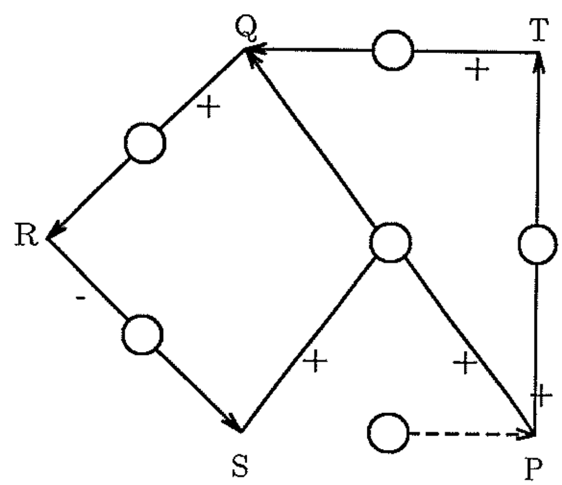

Figure 2.

discuss the special characteristics of the head node and those of the environment. Theorems and selected proofs appear in Section 4.

3.1 Actions of an Internal Node

We will refer to Figure 2, throughout this section, in which the dotted justification arc represents the incoming justification from the IE, making P the head node. When an internal node, such as Q, receives its first message from an antecedent, such as T, it becomes *engaged* and T is considered its *engager*. A message is of the following form:

[message type, antecedent, antecedent status, ancestor list, cycle list].

Each time Q receives a message, it will send messages to its consequences, which is R in this figure, if Q's status or label changes, or if its force list is non-empty. We will discuss the properties of the force list in a later section.

It is obvious that an internal node must have available its justification set, its supporter nodes, its current status, etc, to be a RMS belief node. In order to perform the parallel status assignment computation, an internal node must have the following extra parameters. Each of these parameters and their use will be explained later in more depth.

state:
holds the entire state of belief node prior to the beginning of this computation.
ancestor list:
holds the node's current ancestor list.
possibles list:
holds all of the cycles this node has detected.
force list:
holds all of cycles which have previously been detected as unsatisfiable circularities.
signal wait list:
holds all ordered pairs of the form (cycle, 'U') and/or (cycle,'S') to be sent within the signal to the nodes engager.
detection list:
holds those cycle list for which it has sent or will send an unsatisfiable pair (cycle,'U') in its signal back to its engager.

3.1.1 "NIL sweep" Message

If the message type is "NIL sweep", and T is the engager of Q, then T sends a message:

$$[\text{"NIL sweep"}, \mathbf{T}, \mathbf{NIL}, \phi, \phi].$$

to Q. At which time Q performs the following steps: (1) sets T as its engager, (2) saves its entire state prior to receipt of the message (3) sets its status to NIL, (4) clears its ancestor list, (5) makes any changes to its justification list, invalidating any justifications where T was an antecedent if such justifications were previously valid, (6) sends messages of the "NIL sweep" to its consequences, if Q's status changed because of this message, and (7) signals back to T at any time, when all consequences have signaled to Q, unless Q receives another message to which it must return a signal before sending a signal to T.

By Theorems 7 and 9, Q will never receive a message of another type as long as it is engaged because of a "NIL sweep" message, and the transitive closure of consequences of the head node will be set to NIL before sending another message of a type other than "NIL sweep".

3.1.2 "Label" Message

The "label" message has the same format as the "NIL sweep" message, except the type of message is different, and all statuses being sent will be either IN or OUT.

Any node, such as R, which receives a "label" message from an antecedent, such as Q, will perform the following:

If R's detection list is non-empty, then it has sent a pair, (cyclelist, 'U') in the signal to its previous engager when R became neutral, for each cyclelist that is an element of R's detection list. Upon receiving a label message at this time, R will place new pairs in it's signal wait list of the form (cyclelist, 'S') for all elements of its detection list, and set the detection list to empty. This is necessary for handling circularities that may have looked unsatisfiable at one point during the computation, but may actually be satisfiable once all the labels have been propogated. The elements of the signal wait list will be included in R's signal to its engager. Once R signals its engager, its signal wait list is cleared.

If the cycle list sent within Q's message is non-empty, R must place the cycle list in its force list after it has sent its messages out or has determined that it does not send any further messages. If R determines that it must send messages because of the message it received from Q, R must include the cycle list received in its message to its consequence in the cycle list. This ensures that each member of the cycle becomes aware of its presence on that cycle. If the cycle list is already present in the force list, this cycle list will not be passed in the next messages sent by R. If R determines that it does not send any messages because of Q, then the cycle list is not passed. Below, when the case asks if R's force list is empty, it is asking for the state of the force list prior to R's receipt of the message.

R will then do the following steps if **R itself does not appear in the ancestor list sent by Q.**

Case 1: the message does not cause R to change status and R's force list is empty.

R may signal back at any time to the sender of the message, obeying all rules of signaling. R will not send any further messages to its consequents.

Case 2: the message does not cause R to change status and R's force list is non-empty.

Each element of the force list tells R which possible unsatisifiable circularities it is on. R intersects each element of the force list with its consequences. For each result of the intersection, which is the consequence on the cycle that R is present on, R is forced to send its current status. The force list is then set to empty.

Case 3: the message does cause R to change status.

Since R changes status, it must change its supporter-nodes and ancestors. Then it sends its new status, and ancestor list using another "label" message, to all of its consequences. If R's force list is non-empty, it sets it to empty since those consequences on the force list will receive a message from R anyway.

Internal node R will do the following if **R does appear in the ancestor list sent AND R's possibles list does not contain an element equivalent to the ancestor list R received.** In other words, this is the first time R has detected it is on this specific cycle.

For **Case 1** above, R will not send any messages to any consequents. Since it does not send messages through the cycle, it does not consider seeing a cycle at all.

For **Case 2** above, R is the first node of the cycle it received to detect that this cycle exists. But R may be involved in another cycle which may be an unsatisfiable circularity, therefore R must propagate its current status to the consequents on the cycles that are elements of the force list, but not necessarily to all consequents.

For **Case 3** above, since R must change status under these circumstances, it places the cycle list from the ancestor list Q sent in its possibles list. Then R resets its own ancestor list to itself only. All this is in preparation to detect an unsatisfiable circularity. R then sends out "label" messages to all its consequents, but it includes the detected cycle in the cycle list field of the message to its consequence on the detected cycle. The force list is reset to the incoming cycle.

Internal node R will do following if **R does appear in the ancestor list sent AND R's possibles list does contain an element equivalent to the cycle R received on the ancestor list from Q.**

For **Case 1** above, Since R has no reason to pass any "label" messages to its consequences, it does not detect an unsatisfiable circularity. The cycle is removed from its possibles list.

For **Case 2** above, R is both engaged and has detected its own presence on a cycle. But since its status does not change, R cannot detect an unsatisfiable circularity. R may be involved in other unsatisfiable circularities because its force list is non-empty. Thus R must send out its current status to all consequents that are elements of cycles on the force list, though R does not send out the cycle list it just received in these messages. Then the force list is set to nil and the cycle is removed from R's possibles list.

For **Case 3** above, this will be the second time R is forced to change status due to the same cycle. Therefore R is the first to detect a possible unsatisfiable circularity. It moves the cycle from the list of possibles to the detection list, (deleting it from its possibles list), places this cycle in an ordered pair with 'U' to send in its signal wait list R does not send any messages to any consequences, since it is currently in an inconsistent state. This is upheld even if the force list is non-empty. But since no messages are sent out, the force list remains as it is. Thus R ceases the message passing around the nodes of this particular cycle.

The force list is what ensures that if any node on a cycle receives a message after an unsatisfiable circularity has been detected by some node on the cycle, that the node which detected the circularity and has included it in its signal to its engager, will now include the matching satisfied pair in its signal to its engager. This is a necessary construct, since a node may think it sees an unsatisfiable circularity and send such a signal, because of a delay in the message passing of another node which will satisfy the cycle. Petrie gives an example of this using Figure 2, in which the dotted line represents the incoming justification for P. Thus P is the head node. After P sends out its label messages to Q and T, messages are passed around the cycle Q, R, S until Q detects an unsatisfiable circularity and signals it to P, only to receive a satisfying message from delayed T, which causes Q to signal the matching satisfying pair to T, which propagates it back to P. So no unsatisfiable circularity is present.

3.1.3 "Reset" Message

A third and final message that a node, say Q, may receive is the "reset" message. The "reset" message also has the same format with all of its fields empty except for the type field. This message instructs Q to reset itself to the state which it saved when it received its first "NIL sweep" message. The reset message is sent when unsatisfiable circularity was detected within the graph during the assignment of IN and OUT status to the belief nodes. We leave it as the responsibility of the head node to detect the actual presence of such a circularity as will be presented in the next section.

Instead of leaving the graph in a state of inconsistent labelings, the graph will return to the state it was in prior to the entry of the justification which caused the diffusing computation. It is this part of the computation which is incomplete, in that we do not solve the labeling problem, but attempt to avoid it.

The step an internal node follows when it receives the "reset" message is to simply reset its state and pass the message to its consequents.

3.2 Actions of the Environment and Head Node

The head node is the consequent of the incoming justification. This node is engaged by the environment. Every internal node will have the algorithm available for execution if it becomes the head node, but only one such node will execute the code during a single diffusing computation. The justification will be added to the head node's justification set and the appropriate supporter-nodes identified. Computation proceeds only if the justification is valid and the head node changes status from OUT to IN. The first set of messages the head node initiates are the "NIL sweep" messages. The head node will only be set to NIL if it is involved in some cycle present in its transitive closure of consequences, as in Figure 1, where the dotted line represents an incoming justification. Theorem 7 shows that eventually the head node will receive all signals back from its consequences after sending the "NIL sweep" message. After all signals are received, the head node initiates the "label" messages. It is obvious that every node that receives a "NIL sweep" message will eventually receive a "label" message. If the head node has the label of NIL, it must check its justifications to change its own status, from NIL to IN or OUT before sending the first "label" message. In Fig 1. the entering justification became invalid, so the head node, P, assigns itself the status of OUT, and sends its new status in the "label" messages to Q.

Since the head node is also an internal node, it may detect an unsatisfiable circularity which involves it. If the head node detects an unsatisfiable circularity in which it is involved, it saves this detection and the cycle list until all signals are received. Once the head node receives all signals from its consequences, it performs a matching between the pairs (cycle, 'U') and (cycle, 'S') propagated through the signals received, in an attempt to match every pair with a 'U' to a pair with the same cycle list and an 'S'. By theorem 5, the existence of one pair with a 'U' without a match to a pair with an 'S' means there exists an unsatisfiable circularity with the nodes in the unmatched cycle list. Once such a circularity is detected, the head node will send out the "reset" messages to all its consequences. By Theorem 10, eventually P will receive all signals from its consequences. Then P will signal to its engager the environment, which will detect termination. If no unsatisfiable circularity can be detected, P will not issue the "reset" message, therefore not sending any messages to its consequents. As above, it will eventually signal its engager, the environment.

4 Proof of Correctness

This section presents arguments for the correctness of the algorithm for parallel status assignment. Previously we defined a cycle to be a circularity of status assignments, i.e. the status of each node within this circularity actually depends on itself, or a node is its own supporter. A node N detects a cycle C if N sees itself on the ancestor list sent in a message from an antecedent A and the status of A causes N to change status. A node is a *detector* of the cycle if the node detects the cycle. A node is no longer a detector of a cycle if the cycle has been *broken*. Breaking a cycle occurs when a node in the cycle becomes supported by a node outside that cycle. A node N detects an unsatisfiable circularity C, if N is a detector of the cycle C', that matches and at any point prior to C' being broken, N sees itself on the ancestor list sent in a message from the same antecedent as in C', such that N must change status. The definitions of detector and broken unsatisfiable circularity are the same as that for a cycle. The following lemmas are necessary.

Lemma 1 *The detector of an unbroken cycle is unique.*

It is obvious by the definitions above that at least one node detects the cycle.

Let C be a cycle consisting of nodes $N_1, N_2, ..., N_k$ in which two nodes N_i and N_j are detectors. One of these nodes must have detected the cycle first, since each is in the ancestor list of the other. Assume it is N_i, then N_j is in the transitive closure of consequences of N_i. But when N_i detects C, it clears its own ancestor list of all nodes except itself. Thus N_j cannot detect C. Therefore, the detector is unique.

Lemma 2 *The detector of an unbroken unsatisfiable circularity is unique, and this node is the unique detector of the unbroken cycle corresponding to this unsatisfiable circularity.*

If an unsatisfiable circularity exists, by the definitions above, at least one node on the graph will detect it. We can use the same argument as in Lemma 1, to show that only one node will detect it.

Assume that N_i detects cycle C, and N_j detects unsatisfiable circularity C' corresponding to C. Then by definition of detection of an unsatisfiable circularity, C must not have been broken at the time N_j detected it. But N_j must be the same as N_i by Lemma 1, since two nodes cannot detect the same unbroken cycle.

Lemma 3 *If an unsatisfiable circularity C is detected, all nodes on C will have C in their force list.*

Let N be the detector of C. By Lemma 2, N is the detector of the cycle C', corresponding to C. By construction of the algorithm, when N detected C', it alerted all nodes on C' of their presence. In order to be an unsatisfiable circularity, all nodes on C must have changed status as they passed the force list. Otherwise, the unsatisfiable circularity would not have been detected. By Lemma 2, when N detects C, C' was not broken. Thus, all nodes on C will have C in their force list.

Lemma 4 *Let C be an unsatisfiable circularity. If any node on C receives a message then all nodes on C will receive a message.*

By definition, any node on C receiving a message results in the breaking of C. By Lemma 3, all nodes on C have C in their respective force lists.

Let A be the node which receives the message. Two cases are possible. If A changes status due to this message, then it passes a message to all of its consequences. Otherwise, it will still pass a message to its consequent on C, since C is in its force list. This is the same procedure for all nodes on C. Since all nodes on C must pass a message to their consequent on C, all nodes will receive a message.

Lemma 5 *Let C be an unsatisfiable circularity with N as its detector. Another node on C cannot detect C as a cycle unless N has received a message resulting from the breaking of C.*

Let A be a node on C that receives a message which causes the breaking of C. By Lemma 4, all nodes on C must receive and send at least one message to their consequent on C. Let S be the antecedent of A on C. Then S is the last node on C to send a message because of C in its force list. Two cases are possible when S sends it message to A. If A's status does not change, no further messages are sent by A, i.e. a cycle is not detected. Otherwise, only if A is in the ancestor list of S, i.e S did not get support from outside C before sending this message, will A detect cycle C. But N has already received its message, and therefore is no longer the detector of C.

Corollary 1 *Let C be an unsatisfiable circularity with N as its detector. Another node on C cannot detect C as an unsatisfiable circularity unless N has received a message resulting from the breaking of C.*

By Lemmas 1 and 2, a node must detector of cycle C, before it can be the detector of unsatisfiable circularity C.

Lemma 6 *Let C be an unsatisfiable circularity and N be its detector. The breaking of C results in N sending the pair (C,'S') in its signal to its engager.*

By Lemma 3, the breaking of C results in N receiving a message. By construction of the algorithm, because N is the detector of C, N has either sent a pair (C, 'U') or will send this pair to its engager. Also by construction, upon receiving a message after detecting C, N will prepare to send the pair (C,'S') to its engager. Thus the lemma holds.

Remark: It is possible for the detecting node to receive a new message in between detecting the circularity and finally signaling to its engager. This does not effect the theorem, since the detecting node stores the pair in its signal wait list. We can assume the node knows which signals it has sent and which are just there in case a new message is received. If the first pair has not been sent before the arrival of the new message, then both pairs (C,'U') and (C,'S') will be sent through the same signal.

Theorem 4 *If an unsatisfiable circularity remains unbroken the message passing within the circularity will halt.*

Let C be the unsatisfiable circularity, and N be its detector. Part of detecting C, involves N changing status. But by construction of the algorithm, N will not send out any messages when it has detected C. Since N ceases all message sending, and none of the rules in section 2 have been violated by N ceasing, N will eventually enter the neutral state. If C remains unsatisfiable, i.e. C is not broken, then message passing around C will not occur during the rest of the computation. By Lemmas 4, and 6, if any other messages are passed along C, N will eventually signal that C is satisfied. If C is satisfiable then all rules are maintained and N will eventually reach the neutral state. Since these are the only two cases, the theorem holds.

Theorem 5 *Let C be a cycle and P bethe head node. Given P has received all signals corresponding to all "label" messages it has sent out. If the pair (C,'U') is propagated to the head node and no pair (C,'S') is propagated to the head node, then C is ultimately unsatisfiable.*

P cannot match the pair (C,'U') with a pair (C, 'S'). Since (C,'U') has reached P, by construction of the algorithm, there must exist some node N which detected C to be an unsatisfiable circularity. There is matching (C,'S') sent to P by N, so by Lemma 6, N cannot have received another message after the detection of C. By Lemma 4, no nodes on C received any messages. Thus all nodes on C must still be in a state of inconsistency. Therefore, an unsatisfiable circularity exists in the graph.

Theorem 6 *Let C be a cycle. If for every pair (C,'U'), there is a matching pair (C,'S') propagated to the head node, then no unsatisfiable circularity exists.*

Given a matching pair (C,'U'), and (C,'S'), for every C, By Lemma 6, every node which detected some C, also received a message after detection occured. By Lemma 4, each node on the cycle received at least one more message. If the circularity was still unsatisfiable, there would be a node on the cycle which would detect it, creating a pair (C, 'U') with no match. Thus, the circularity must have been satisfied when this later message was received.

Theorem 7 *There will be no overlapping between the "NIL sweep" and the labeling algorithm.*

To show this, we must show that all signals from "NIL sweep" messages eventually reach the head node before any "label" messages are sent. Since there is only one status involved in performing the "NIL sweep", there can be no unsatisfiable circularities. Even stronger, no node will detect any cycle because after the first change in status to NIL, the node will not change again when another "NIL sweep" message is received. We assume all node obey the rules in section 2. Therefore when the head node receives all signals corresponding to the "NIL sweep" messages it sent, all nodes in its transitive closure of consequences will be in the neutral state.

Theorem 8 *There will be no overlapping among the labeling computation and the reset computation.*

By definition of the head node, it will only send out reset messages if an unsatisfiable circularity exists within the graph. The head node can only determine this state in the graph after all of its signals have been received from its successors. By Theorem 4, since message passing withing an unsatisfiable circularity ceases before the detector signals its engager, and we assume all nodes obey rules in section 2, when the head node receives all signals corresponding to the "Label" messages it sent, all nodes in its transitive closure of consequences will be in the neutral state.

Theorem 9 *There will be no overlapping of the "NIL sweep" and the reset computation.*

This theorem is a direct consequence of the previous two.

Theorem 10 *If the reset computation is necessary, the head node will receive all signals corresponding to its reset messages, resulting in termination of diffusing computation.*

We have shown that eventually the head node must receive all signals back from the labeling message (Theorems 4 and 9). Only in the case where the head node has determined there exists an unsatisfiable circularity in the graph will it begin the reset message. By the same reasons applied to the "NIL sweep" message in Theorem 8, the head node will receive all signals from its reset messages, since the reset message causes a node to change status at most once when it resets its state, and this message causes nothing to be sent back with the signals. Thus the head node will eventually signal its engager, the environment, resulting in termination.

4.0.1 Consistent and Well-founded labeling

The algorithm presented in Section 3 dealt primarily with termination in the presence of unsatisfiable circularities. There are additional specifications of each internal node that are necessary to ensure a consistent and well-founded labeling if one exists. Such specifications do not interfere with the use of diffusing computation in the algorithm, but only add an additional condition under which a node may change status. In this section we will present the remaining node specifications and proofs that a consistent and well-founded labeling can be computed if one exists. The following definitions are necessary for this section. The **affected-consequences** of a node are those consequences of the node which contain the node in their set of supporting-nodes. The set of **repercussions** of a node is the transitive closure of the affected-consequences of the node. The set of **believed-repercussions** is a subset of repercussions of a node in which all the affected-consequences of the node are labeled IN.

A RMS network is consistent when each node is assigned a status of IN if and only if it has at least one valid justification, and OUT otherwise. A RMS network is well-founded if no node appears in its own believed-repercussions.

CONSISTENT

We require by definition that the operation of each individual node to determine its status from the definition of a consistent network. Consider only the case in which no unsatisfiable circularities are present, otherwise no consistent labeling of the network can be found. Each node will determine its own consistent status, and pass its status each time it changes. It has been previously shown that the algorithm will eventually terminate. Thus each individual node will eventually reach its neutral state with a consistent label, making the whole network consistent.

Well-foundedness

The case of concern in well-foundedness is when a consistent labeling is established but is not well-founded. This case arises when a cycle exists on which a node is IN and its antecedent on the cycle is the node's supporter. This can be seen in example 4.

In order to maintain a well-founded state, each node must be reponsible for performing some internal bookkeeping on its supporter-nodes. When a node receives a message which entitles it to have IN status but whose ancestor list contains the node on a cycle, then the antecedent which sent the message can never be a supporter of this node as long as its ancestor list has the node on a cycle. As the node receives messages or searches through its justifications for new support, it checks the ancestor lists of the antecedents. The node will not change from OUT to IN if it has no justifications with antecedents, all of which do not contain the node in their ancestor list. If a valid support is removed from a node which allowed it IN status, the node will become OUT if it has no justifications with antecedents all of which do not contain the node in their ancestor list. Thus, we can define the well-foundedness of node to be one in which none of its supporter-nodes contains the node in its ancestor list. If every node is well-founded then the graph of nodes will also be well-founded. By the above restriction on a node having IN-status, it is obvious that every node will be well-founded.

5 Discussion and Conclusion

If the graph formed by the transitive closure of consequences of the head node is acyclic, then all rules obviously hold. We have shown that in the presence of unsatisfiable circularities, all types of message passing and signalling will cease. We have also shown that there is no overlapping of messages, i.e. that each set of messages eventually results in the head node receiving all signals corresponding to the that set of messages it sent out. After receiving all the signals corresponding to "label" messages, the head node can determine whether or not the labeling computation halted with any unsatisfiable circularities remaining in the graph. If so, the head node will initiate a reset computation so that the network returns to a consistent state.

Each node in the graph will either settle on a status assignment or belong to an unsatisfiable circularity. Since there are only a finite number of nodes, each node has a finite number of consequences and there are only a finite number of cycles possible. Therefore, the number of messages sent out by each node due to its status changing or force list is finite. The engager of a node is always signalled last before the node enters the neutral state. Since none of the conditions for termination detection for diffusing computation are violated, this algorithm will terminate.

Other sequential algorithms have been developed for the status assignment problem, such as [6] [1] [8], some of which restrict the presence of "odd loops" in the graph in order to prove that a consistent and well-founded labeling exists. This may be too tight a restriction in the graph, and we believe that each circularity should be dealt with individually when discovered, though this limits our proof of consistency and well-foundedness to those cases when no unsatisfiable circularity exists. Further research includes locking mechanisms so that multiple justifications can be sent by the IE simultaneously to initiate multiple diffusing computations and the handling of contradictions in parallel. [5]

Acknowledgments: Thanks go to Charles Petrie for his initiation of the problem and discussion. We would also like to thank the CICS group for their discussion and readings of this text.

References

[1] Eugene Charniak, Christopher K. Riesbeck, and Drew V. McDermott. *Artificial Intelligence Programming.* Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1980.

[2] E.W. Dijkstra and C.S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1), 1980.

[3] Jon Doyle. A truth maintenance system. *AI*, 12(3):231–272, nov 1979.

[4] Jon Doyle. Some theories of reasoned assumptions, an essay in rational psychology. Technical Report CMU-CS-83-125, Carnegie-Mellon University, May 1983.

[5] R.M. Fulcomer and W.E. Ball. Towards a fully parallel reason maintenance system. In *The 2nd Int'l Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*, 1989.

[6] James W. Goodwin. Watson: A dependency directed inference system. In *Proceedings of Non-Monotonic Reasoning Workshop*, pages 103–114. American Association for AI, oct 1984.

[7] C. J. Petrie. A diffusing computation for truth maintenance. In *Proceedings of the IEEE International Conference on Parallel Processing*, pages 691–695, 1986.

[8] D. Rusinoff. An algorithm for truth maintenance. Technical Report MCC AI-062-85, Microelectronics and Computer Technology Corp., April 1985.