Report Number: WUCS-89-03

1989-01-01

Towards a Fully Parallel Reason Maintenance System

Rosanne M. Fulcomer and William E. Ball

A truth maintenance system (TMS) is an AI system used to monitor consistency of information in a knowledge base. A TMS may be necessary when non-monotonic reasoning is used since incorrect assumptions can lead to contradictory conclusions. The Reason Maintenance System (RMS), a specific TMS first described by Doyle [5],[6], is used along with an inference engine (IE), or problem solver, to maintain a consistent set of beliefs and inferences. We have developed a parallel version of the RMS for correctly assigning IN or OUT states to each believe node [7]. This algorithm uses diffusing computation [4] to assign... **Read complete abstract on page 2.**

Towards a Fully Parallel Reason Maintenance System

Rosanne M. Fulcomer and William E. Ball

Complete Abstract:

A truth maintenance system (TMS) is an AI system used to monitor consistency of information in a knowledge base. A TMS may be necessary when non-monotonic reasoning is used since incorrect assumptions can lead to contradictory conclusions. The Reason Maintenance System (RMS), a specific TMS first described by Doyle [5],[6], is used along with an inference engine (IE), or problem solver, to maintain a consistent set of beliefs and inferences. We have developed a parallel version of the RMS for correctly assigning IN or OUT states to each believe node [7]. This algorithm uses diffusing computation [4] to assign the status to a node. In this paper we will further parallelize the RMS by superimposing a locking mechanism on the RMS to have simultaneous status assignment computations performed. Also, we will address how contradiction handling can be executed in parallel and the effect on the RMS when a parallel contradiction handler is incorporated.

TOWARDS A FULLY PARALLEL REASON MAINTENANCE SYSTEM

Rosanne M. Fulcomer
William E. Ball

WUCS-89-03

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 2nd International Conference on Industrial & Engineering Applications of Artificial Intelligence & Expert Systems, 1989.

This work was supported by the Center for Intelligent Computing Systems at Washington University.

Toward a Fully Parallel Reason Maintenance System*

Rosanne M. Fulcomer
William E. Ball

Department of Computer Science
Washington University
One Brookings Drive
St. Louis, MO 61630

Abstract

A truth maintenance system (TMS) is an AI system used to monitor consistency of information in a knowledge base. A TMS may be necessary when non-monotonic reasoning is used since incorrect assumptions can lead to contradictory conclusions. The Reason Maintenance System (RMS), a specific TMS first described by Doyle [5],[6], is used along with an inference engine (IE), or problem solver, to maintain a consistent set of beliefs and inferences. We have developed a parallel version of the RMS for correctly assigning IN or OUT status to each belief node [7]. This algorithm uses diffusing computation [4] to assign the status to a node. In this paper we will further parallelize the RMS by superimposing a locking mechanism on the RMS to have simultaneous status assignment computations performed. Also, we will address how contradiction handling can be executed in parallel and the effect on the RMS when a parallel contradiction handler is incorporated.

1 Introduction

The Reason Maintenance System [5] [6] is a specific type of truth maintenance system. Truth maintenance systems are an important AI technology because they can reason without complete information using assumptions and can revise beliefs as new information becomes known. Belief revision is important in maintaining consistency among the beliefs represented in the truth maintenance system. Such a system can also maintain dependencies between information and give reasons as to why certain information is believed true. Performing this type of sequential reasoning is computationally expensive.

In the next section, we will give a brief overview of the RMS, which will include how status assignments and contradictions are handled. The problems of unsatisfiable circularities, nogood nodes, and conditional proof justifications will be addressed. Also, in this section, we briefly describe the solution developed by Petrie [1] to avoid nogood nodes and conditional proof justifications. In Section 3, an overview of a completed algorithm [8] which correctly performs status assignment computation in parallel will be presented. In Section 4, we will describe a way to increase the parallelism of the entire system. Our proposed solution to handle contradictions in parallel and its incorporation into the whole system will be discussed in Section 5. Section 6 gives discussion of related future work.

2 The Sequential Reason Maintenance System

2.1 Status Assignment in RMS

Inferences are passed to the RMS, which creates a node for each belief and maintains the dependencies between these beliefs. Each node in the RMS is assigned a status, or label, 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**, which give reasons for the status or label of the belief. Each justification, termed a support-list (SL) 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. For each justification, there will

*Supported by the Center for Intelligent Computing Systems, Washington University, St. Louis, MO.

be only a single consequence. An **assumption** is a belief having a supporting justification with a non-empty OUTSET. A node which has an empty justification set is always OUT.

We illustrate this labeling in Figure 1. Each letter is a belief or node in the RMS. A justification is pictured by the circle, in which edges from the letter to the circle with a "+" mean that letter is a belief in the INSET of the justification and those with a "-" indicate the letter is a belief in the OUTSET of the justification. The arrow from the circle to a letter indicates the letter is the consequent of that justification. Therefore, in Figure 1, A is in the INSET of the only justification for B and in the OUTSET of the only justification for D. C is in the INSET of the only justification for D. For B to be IN, A must be IN and for D to be IN, A must be OUT and C must be IN.

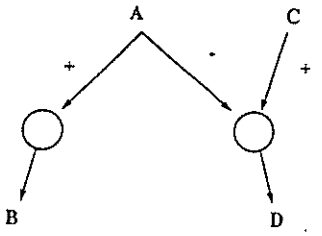

Figure 1: Example of Structure

Problems arise in Doyle's status assignment computation due to **unsatisfiable circularities**, which cause infinite looping. An unsatisfiable circularity is shown in Figure 2, in which B can be IN only if it is also OUT.

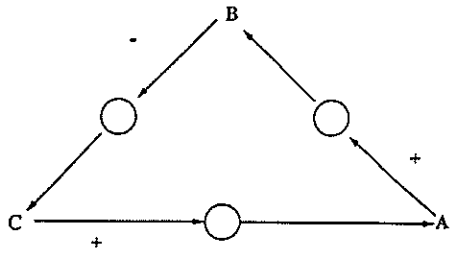

Figure 2: Unsatisfiable Circularity

In order to avoid the possibility of encountering unsatisfiable circularities, some have restricted the RMS to use only data without "odd loops" [3] [9]. An "odd loop" is characterized by an odd number of OUTSETS in a cycle, e.g. there is one "minus" present among all the arcs in the cycle in Figure 2. We choose to not have this restriction.

2.2 Contradiction Handling in the RMS

If the RMS receives information that certain nodes labeled IN cause a contradiction, it will create a contradiction node representing this information. If this contradiction node remains IN after all status assignment has been completed, dependency-directed backtracking must be performed to find the set of maximal assumptions in the foundations of those nodes causing the contradiction. Doyle's algorithm creates what is called a nogood node, which references those nodes causing the contradiction. The nogood node is justified by a conditional proof (CP) justification, which indicates which contradiction node will be IN due to the status of certain nodes. To resolve the contradiction, a new justification is created, which justifies an OUT belief which caused one of the assumptions to be IN, so that the contradiction node can be labeled OUT. The belief chosen from the maximal assumption set is called the culprit, A chosen node in the OUTSET of the supporting justification of the culprit is called the elective. It is the elective that is the consequent of the newly created justification which causes the contradiction node to have OUT status. We illustrate this in the following example.

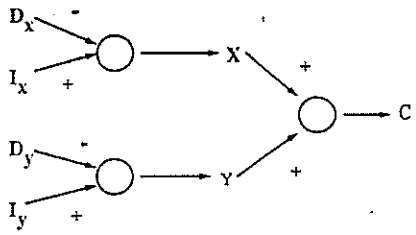

Figure 3a: Notification of Contradiction

New Node: NOGOOD-X-Y

with CP-Justification: Consequent = C
INhypothesis = {X,Y}
OUThypothesis = {}
Status = IN

New Justification: D_j

with SL-Justification: INSET = {NOGOOD-X-Y,Y}
OUTSET = $\{D_x - \{D_j\}\}$
Status = IN

Figure 3b: Contradiction Resolution

Looking at Figure 3, we assume the RMS has just been notified that nodes X and Y cause a contradiction. Let C be the contradiction node created. Using dependency-directed backtracking, the RMS finds nodes X and Y are the assumptions first encountered in the foundations of C. The set {X,Y} is called the maximal assumption set. From this set, a culprit is chosen, say X, with OUTSET D_X, i.e X is an assumption because its supporting justification has a nonempty OUTSET. One element, D_j is selected from D_X and is called the elective. Then D_j is the node that is justified with a new justification causing it to be IN. Because of D_j's new status, C will be assigned an OUT status. This justification has as its INSET the created nogood node unioned with {Y} (or the rest of the assumptions in the maximal assumption set without the culprit.) The OUTSET of this justification contains the set $D_X - D_j$ (or the rest of the OUTSET of X without the elective.)

2.3 Revision to RMS Contradiction Handling

The use of nogood nodes and conditional proof justifications when handling contradictions causes problems in parallelizing this mechanism. Particularly, the representations and creations of both the nogood node and the conditional proof are difficult. As discussed later, our parallel status assignment algorithms uses a representation for SL justifications that is not easily converted to CP justifications. Nogood nodes pose the problem of being created dynamically in the middle of the contradiction resolution. This will become more clear in the following sections.

Both the nogood node and CP justification create problems in the sequential system, i.e. the CP justification must be emulated by a SL justification for status assignment computation which is expensive, and CP justifications restrict the instances in which the nodes they support can be IN [5]. Though other systems using truth maintenance have attempted in correct this problem, we have found that the changes that Petrie proposes [1] [2] are the closest to what we expect from the contradiction handler and these changes also aid the parallelizing of the contradiction handler. In [1] can be found a brief review of those other systems mentioned.

Petrie's primary focus is to extend the capability of justifications to provide simple explanations for the belief of certain information, especially when the resolution of a contradiction contributes to the belief of that information. His approach no longer makes use of either the nogood node or the CP justification. Instead an SL justification is created for an existing node which will resolve a contradiction. The definition for the culprit and elective are the same as earlier in the paper. But, Petrie must redefine the term assumption to be a node with a supporting justificaton with an OUTSET containing at least one element which is not a contradiction.

Let C be the contradiction node. Dependency-directed backtracking yeilds A, the set of maximal assumptions. A belief P is chosen to be the culprit. Let D be the OUTSET of the supporting justification of P. An elective $D_j \in$ D is chosen. To justify D_j to be IN, a new justification is created. The INSET of this justification is the set consisting of, $(A - \{P\}) \bigcup I \bigcup BI$. The OUTSET is the set, $\{C\} \bigcup BO \bigcup (D - \{D_j\})$, where I is the INSET of P and BI and BO are respectively the IN and OUT nodes required by Doyle for the SL justification to implement the CP justification.

BI and BO are obtained as follows. Let F be the set of all nodes in the foundations of C. Let R the set of nodes in the repercussions of the elements of A (where the repercussions of a node are formed by taking the transitive closure of consequences for which the node is a supporter.) Let T be the set antecedents of C, and TR be the set of antecedents of the nodes in R. Then some belief, N $\in$ BI, if N is IN and $N \in \{F \bigcap \neg A \bigcap \neg R \bigcap (T \bigcup TR)\}$. N $\in$ BO if the above holds except N is OUT. Thus, a new justification is created that is not a conditional proof justification to resolve the contradiction.

It is pointed out that this solution may cause an unsatisfiable circularity because of the addition of this new justification for the elective. This will happend if an element of this justification is contained in the transitive closure of consequences of the elective, and if there is now an "odd loop" containing that element and the elective. If both of these conditions are satisfied, the justification cannot remain in the system. A new elective is then chosen. If no new elective exists, a new culprit is selected from the set of maximal assumptions. If no such culprit exists, the contradiction cannot be resolved.

Petrie extends his sequential contradiction handling algorithm [2] to perform resolution without finding the maximal assumption set. He also provides reasons for the following changes in his reasoning mechanism. If an elective of a culprit has no justification, then the above solution is used. But if the elective has justifications that are invalid because of the lack of justification for another set of nodes, i.e. those in the OUTSET of the elective's invalid justification, then it is this set of nodes that should be justified by contradiction resolution. For simplicity, only Petrie's initial changes will be addressed in our parallel algorithm.

3 Parallel Status Assignments

We have developed a parallel version of the RMS for correctly assigning IN or OUT status to each belief node [8]. This algorithm uses the method of diffusing computation [4] to assign the status to a node according to [5]. In this method we consider each belief node to be a separate processor and a justification is represented as a directed arc from antecedent to consequent (also see [10] for the same general mapping.)

Diffusing computation is a method in which computation proceeds by passing messages in a finite, directed graph of processors. A processor is activated only when it receives a message. When a processor is activated, it performs its calculations and sends the result in a message to its successor. Termination can be detected by having processors issue signals as replies to the messages they have received.

The **environment node** is the node which receives the justification from the inference engine (IE) or problem solver and initiates status assignment computation. It does this by sending a message to the consequent of the new justification, which is called the **head node**. The head node and all other beliefs are considered to be **internal nodes**. It is the environment that detects termination of the computation by the receipt of a signal from the head node. The head node issues this signal to the environment when it has received signals to all messages which it has sent to its consequents. Information can also be passed within these signals. We make the restriction, that everything received in a signal eventually propagates to the head node. A successor node (referring to a directed graph) is considered **engaged** if it has received a message from a predecessor. A node that is not engaged is in its **neutral state**. The first predecessor that sends a message to a successor is called the **engager** of that successor. Signalling from a successor to a predecessor is restricted so that the engager of the successor receives the last signal from the successor before the successor goes into its neutral state. A node may go from neutral to engaged several times during a single computation.

Status assignments can easily be performed in parallel using this method except in the presence of an unsatisfiable circularity. Reassignment of status to the belief nodes begins when the IE passes a new justification to the RMS causing an OUT node to change to IN. We require that a node send its status to all of its consequents if it receives a message which changes its status or, upon receipt of a message, it sends its status to those consequents involved with it on a cycle. The first requirement is to compute the status assignments as the sequential RMS does. The second requirement is necessary in such a message passing scheme and will be explained below.

We have solved the problem of stopping the possibly infinite number of status assignments made to each node in the circularity, which caused non-termination or false termination detection of the computation in a first attempt [10], by creating a mechanism which makes the node which detects the circularity be the only node which reports the circularity to its engager. This same node will report the presence of the cycle to the rest of the nodes on the cycle. Each node must know that it may be on an unsatisfiable circularity.

The problem with message passing is that a processor may be delayed in sending its status message. This may cause a satisfiable cycle to look like an unsatisfiable circularity. This is because the delayed processor may change the status assignment of a node on the circularity. This change may satisfy the circularity, leading to a consistent set of status assignments for the nodes on the circularity. If this occurs, the node which first detected the circularity must be notified so that it can retract its previous detection, by reporting the satisfiability of the circularity to its engager. Notifying the detector node can only be accomplished if all nodes on the circularity are aware that this detector node exists. We force nodes to be aware that they are on a cycle. Then, if they receive another message, and follow the second requirement above, eventually the detector of the unsatisfiable circularity will receive a message, causing it to report that the cycle has now been satisfied. Of course, this new message may not cause the cycle to be satisfied, even though a report is given. But, eventually, its unsatisfiable nature will be detected again and reported. The head node will receive these reports and from them can determine if an unsatisfiable circularity exists.

When this circularity problem arises, it is an indication that there is a problem with the beliefs and dependencies among the beliefs within the inference engine. If this occurs, all nodes involved in this diffusing computation will be reset to the state they were in prior to the computation, and the justification and belief nodes involved in the circularity will be passed back to the IE for handling. The full algorithm and its proof of correctness can be found in [7].

4 Status Computation Using Multiple Environments

In the above solution, only a single justification can be processed at a time. In this section we will discuss how multiple environment nodes can be used to allow disjoint status assignment computations to be performed simultaneously.

To achieve maximum parallelism, multiple justifications must be handled in parallel. This requires the

RMS to process justifications as they arrive from the IE, allowing multiple, separate status assignment computations to occur simultaneously.

Instead of a single environment node, we propose to allow some static number of environment nodes to be present in the RMS, each of which can process an incoming justification and initiate a status assignment computation. In Figure 4, we are given that there are K environment nodes, $e_1, e_2, ..., e_k$. Each of these environment nodes can communicate with each belief node.

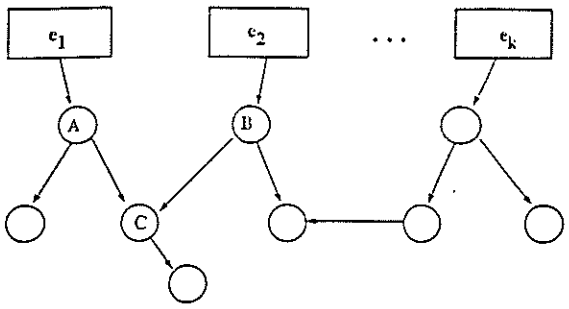

Figure 4: Multiple Environment Nodes

To devise a scheme which divides the nodes among the environment nodes so that little interference occurs is much too costly, and would require the environment nodes to communicate with each other, which we would like to avoid. Also, since new justifications can enter, creating new dependencies, a division scheme would have to be dynamic in nature. But in the absence of such a scheme a locking mechanism must be utilized.

The RMS will be restricted to report to the IE or to give explanations only at quiesence or steady state which is achieved when all environment nodes are not active. An environment node is considered active when it has initiated computation which has not terminated. Any inactive environment node may receive information from the IE or problems solver and begin computation on that information.

To perform multiple status assignment computations, a unique tag will be assigned to each environment node. When an environment node initiates status assignment computation by informing the head node of its new justification, its tag is included in the message. We define a node as being *engaged under environment T* if this node was engaged by a message containing the unique tag for environment T. Then, as long as a node is engaged under a certain environment, it passes that environment's tag in all of its messages and cannot accept messages from nodes engaged under a different environment.

Before we proceed to the details of the locking mechanism, we will illustrate the general restrictions on communication with Figure 4. Let node B be engaged under an environment with tag e_2, and node A be engaged under an environment with tag e_1. If node A sends its status message to node C before node B (assuming that messages arrive in the same order as they are sent), then node C will be engaged under environment e_1. Node C is now restricted from receiving any message from any sender with a different environment tag. If B attempts to send its status to C, its message will not be received by C, and B will be blocked from completing its computation until C receives its message. B is only blocked from completion, but can still send its status messages to consequences not engaged under different environments.

4.1 Locking Mechanism

In this section we examine the problem of the above proposal and show what extra constraints must be added to achieve a correct solution.

We are given a static number of environments. The purpose of each individual environment is to perform a status assignment computation in a RMS dependency graph. Cycles are allowed in the graph, and we know that if status assignment is performed according to [8], the computation by a single environment will terminate. We consider this to be a single status assignment computation. It is possible for multiple environments to want to engage the same node. This overlap can cause two problems: deadlock may occur and consistency of the dependency graph may be violated. We address these problems in the above order.

In a single status assignment computation, there are three types of messages sent out by the head node: the "NIL" sweep, "label", and "reset" messages. Unless drastic changes are made in the status computation itself, no message type from one head node can overlap a message type from another head node. Since during the "NIL" sweep, the engagement tree under an environment must consist of all reachable nodes from that head node, all such nodes must be locked before the "NIL" sweep messages are sent. These nodes must remain locked throughout the computation of status assignment. Once the receives its final signal the nodes can be unlocked. Thus, the only way that two status computations can be performed simultaneously is if there is no link between any of the nodes in each computation.

In the following solution, performing the locking requires the environment nodes to have some type of ordering. We will allow the ordering to be determined by the IE passing the justification to an environment and giving the environment a tag, increased with respect to the previous tags it has given. Then the ordering on the environments will be determined by the environment's tag. If multiple environments are competing to lock the same node, the environment with the smallest tag

is considered the oldest and receives the lock. Along with showing that the problems above cannot occur, we must be able to show that starvation of an environment cannot occur under this ordering.

Reservation requests are sent out before locks. Each parent node must receive acknowledgments that it has reservations from all of its children before locking them. Once all reservations are received from the child, the parent can send out locks.

We will define three new types of messages "reserve", "lock", and "cancel". A parent sends out the reserve messages and waits for a response of "yes", the reservation is complete or "no", the child cannot be reserved. A "no" response requires the parent to send cancel messages to its children causing them to cancel the reservations they have from their successors under this parent's environment. If a child is reserved/locked under an environment, then the receipt of a message from a parent under the same environment is acknowledged with a "yes". The receipt of a message from a parent under a different environment is always held, even if a message from that parent or another from the different environment exists. A reservation can be changed if a parent from a higher priority environment attempts to reserve the child. If the child has already acknowledge a lower priority environment parent, then the higher priority environment waits instead of cancelling.

The key to the lack of deadlock is that when competition arises, the lower priority environment cancels, while the higher priority waits.

The locks are released by the environment when it detects that computation has terminated.

4.1.1 Starvation

By definition, starvation occurs if there is an environment ready to process an incoming justification, but this environment is never allowed to retain all of its necessary locks.

To show lack of starvation, we must be able to show that an environment that cannot receive its locks at some point will eventually become the environment with the highest priority, and having this priority will allow it to eventually receive all of its locks.

We are given a static number of environments, say E. Let environment e_i, have the number i as its tag, and let $\leq$ be the ordering of the tags, such that if i,j $\in$ N, and i $\leq$ j then, i has a higher priority that j. At all times there are a finite number of environments with lower priority that e_i and a finite number of environments with higher priority than e_i. Assume it is possible for e_i to be starved from retaining its necessary locks. By construction of the algorithm, those environments with greater priority competing with e_i require e_i to cancel its previous reservations. Those environments with less priority competing with e_i allow e_i to wait and get the reservation as soon as it is available. Thus, eventually e_i will become the environment with the highest priority.

Since all other environments have lower priority, when e_i sends out requests for lock reservations, any beliefs not locked immediately become reserved for e_i and cannot be taken away. As for those beliefs already locked, e_i will wait to receive them as soon as they are released. Thus, no starvation can occur.

4.1.2 Consistency

To show consistency, we need to show that if we give a belief two pieces of knowledge and the belief determines its status to be IN(OUT) after this combined knowledge has been processed, then if we present the knowledge in any sequential order, the belief will still determine its status to be IN(OUT).

Assume that the status of the belief as described above is IN. Let the two pieces of knowledge be called K_i and K_j. Let K_i be the first piece of knowledge presented and K_j be the second piece of knowledge presented. Then no matter what status the belief determines after K_i is processed, the belief determines its status as IN after K_j is processed. We must show that we can change the order of K_i and K_j and still receive the same status.

Let the belief be called B. Assume B has a status of IN before both K_i and K_j are processed. (1) If B has a status of IN after K_i is processed, then K_i and K_j can be processed in any order. (2) If B has a status of OUT after K_i, then K_i must have caused all the valid justifications of B to become invalid. If K_j is processed, it must validate at least one of those justifications. The justification that it validates to support B cannot be made invalid by K_i, otherwise it would never have been valid by K_j. Thus, K_i and K_j can be processed in any order.

Assuming the same sequential ordering, let B has a status of OUT before both K_i and K_j are processed. If B has a status of IN after K_i is processed, then K_i and K_j can be processed in any order. If B's status remains OUT after K_i is processed, then any changes K_i made, did not cause any justifications to become validd. But K_j caused some justification to become valid, either by the changes it made alone, or the combined changes it made with K_i. Thus switching the ordering would not change the ending status. Similar arguments apply with B having a final status of OUT.

In the case of unsatisfiable circularities, the new knowledge causing the unsatisfiability is thrown out, and the net is set back to its original state prior to the processing of that knowledge. Thus, we can simply

view the problem as one in which that knowledge never existed.

4.1.3 Deadlock

Deadlock will occur if there exists a cycle of environments waiting on each other for their necessary locks.

Given that there are N environments and that we have a cycle in which some number m, $2 \leq m \leq N$, environments are waiting on each other. Because of the total ordering placed on the priorities of the environments, there is one environment with the highest priority and one environment with the lowest priority. For convenience, we can relabel the environments with tags from 1 to m, with the environment whose tag is 1 having the highest priority. Now we can imagine a cycle of environments (with their tag as a subscript) as follows: e_1, e_i, ... , e_j, e_1, $2 \leq i,j \leq m$ and $i \neq j$, in which each environment is waiting to lock a belief(or beliefs) that its right neighbor has. The environment e_1 has the highest priority. By construction of the algorithm, if e_j attempts to reserve a belief that e_1 has reserved, it will fail and will cancel all other beliefs that it has reserved. The environment e_j has environment e_{j-1} waiting on it so e_{j-1} must have a higher priority than e_j. So e_{j-1} can reserve all of its locks, which breaks the cycle.

5 Handling Contradictions in Parallel

As stated earlier, using conditional proof justification and nogood nodes poses problems for the both sequential algorithms and for parallelizing Doyle's contradiction handling mechanism. Thus we will attempt to parallelized Petrie's contradiction resolution algorithm and incorporate it into the parallel RMS. To begin, we must extend the capabilities of the environment node. At the same time, we wish to have the environment node maintain the same role in status assignment computation.

The IE sends the justification for a contradiction node, which does not exist in the RMS. Upon receiving a justification for a contradiction the environment creates a new contradiction node and supplies it with its justification. This contradiction node is also an internal node. At this point the environment must initiate dependency-directed backtracking.

5.1 Finding the Maximal Assumptions

It is possible to find this set of assumptions by using a parallel graph search strategy initiated by the environment, starting at the contradiction node and continuing (against the direction of the arcs) through the INSETs of the supporting justifications of each node, until an assumption is reached (using Petrie's redefinition). Once an assumption is reached, the search along that path terminates and the assumptions and its OUTSET are returned up the path.

Diffusing computation can be used to perform the assumption search, in which each node that receives a message determines if it is an assumption. If the node is not an assumption, it sends messages to the members of its INSET. If it is an assumption, it signals to its engager that it is an assumption and returns itself and its OUTSET in the signal. All nodes that are not assumptions will collect the pairs of assumptions and their OUTSET, union them together as they are received in signals, and send them in their signals to their engager. Eventually the contradiction node will send the set of pairs which contain members of the maximal assumption set and their OUTSETS to the environment. Though the dependency directed backtracking will terminate, the environment is still occupied with the handling of the contradiction.

5.2 Choosing a Culprit and Justifying an Elective

Once the environment node receives the set of assumptions, it randomly chooses one to be the culprit, and from the culprit's OUTSET it randomly chooses an elective. Next, a new justification is created to cause the elective to be IN, according to the specifications given in an earlier section. Since our parallel status assignment terminates in the presence of unsatisfiable circularities, and resets the nodes to their previous status, we can skip the step to check the transitive closure of consequences of the elective, and immediately perform parallel status assignment with the elective as the head node in order to change the status of the contradiction node to OUT. Then if the new justification causes an unsatisfiable circularity, it will be ignored, and the environment will be free to choose another elective if possible. If no new electives or culprit with other electives can be chosen, the contradiction cannot be resolved and the database remains in a contradictory state.

5.3 Contradiction Handling Using Multiple Environments

Given the algorithms necessary to ensure that simultaneous status assignment can occur, we hope to extend them to allow simultaneous contradiction handling and intermixing of status assignment and contradiction handling. Problems arise in simultaneous contradiction handling because the environment performs most of the computation in the contradiction handling and initiates

different types of computation. Thus, a node is not constantly engaged under an environment and interference can occur yeilding results that are incorrect. We believe that the same or very similar locking mechanism can be utilized if we immediately lock the contradiction node before reserving any other nodes and give the environments handling contradictions an implicit higher priority over those environments only handling status assignments. Then, none of the antecedents in the foundations of the contradictions would be able to be reserved.

6 Discussion and Conclusion

The most obvious problems with this first attempt at parallelizing the contradiction handler is that only two steps can be easily parallelized, i.e. the dependency-directed backtracking and the status assignment initiated by justifying the elective. Also, the gathering of the information for the new justification of the elective is very difficult to perform without repeated passes over the network. These passes can only be done when they do not interfere with other ongoing computation. Because we had hoped to exploit parallelism to a greater degree, we are making a second attempt at parallelizing the contradiction handler which may diverge significantly from the sequential version.

The locking mechanism must also be improved to handle contradictions and simple status assignment computation. Another improvement is to have a different kind of priority ordering of the environments not dictated by the IE, so the at multiple IEs can interact with the RMS or one IE can send justifications simultaneously.

References

[1] C. J. PETRIE, J. Using explicit contradictions to provide explanations in a tms. Tech. Rep. AI/TR-0100-05, Microelectronics and Computer Technology Corporation, November 1985.

[2] C. J. PETRIE, J. Extended contradiction resolution. Tech. Rep. AI-102-86, Microelectronics and Computer Technology Corporation, March 1986.

[3] CHARNIAK, E., RIESBECK, C. K., AND MCDERMOTT, D. V. *Artificial Intelligence Programming.* Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1980.

[4] DIJKSTRA, E., AND SCHOLTEN, C. Termination detection for diffusing computations. *Information Processing Letters 11*, 1 (1980).

[5] DOYLE, J. A truth maintenance system. *AI 12*, 3 (nov 1979), 231–272.

[6] DOYLE, J. Some theories of reasoned assumptions, an essay in rational psychology. Tech. Rep. CMU-CS-83-125, Carnegie-Mellon University, May 1983.

[7] FULCOMER, R., AND BALL, W. Correct parallel status assignments for the reason maintenance system. Tech. Rep. WUCS-88-26, Washington University, 1988. Submitted for publication.

[8] FULCOMER, R., AND BALL, W. Correct parallel status assignments for the reason maintenance system. In *to appear 1989 Int'l Joint Conference on Artificial Intelligence* (August 1989).

[9] GOODWIN, J. W. Watson: A dependency directed inference system. In *Proceedings of Non-Monotonic Reasoning Workshop* (oct 1984), American Association for AI, pp. 103–114.

[10] PETRIE, C. J. A diffusing computation for truth maintenance. In *Proceedings of the IEEE International Conference on Parallel Processing* (1986), pp. 691–695.