An Assumption-based TMS

Johan de Kleer

Intelligent Systems Laboratory, XEROX Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA 94304, U.S.A.

Recommended by Daniel G. Bobrow

ABSTRACT

This paper presents a new view of problem solving motivated by a new kind of truth maintenance system. Unlike previous truth maintenance systems which were based on manipulating justifications, this truth maintenance system is, in addition, based on manipulating assumption sets. As a consequence it is possible to work effectively and efficiently with inconsistent information, context switching is free, and most backtracking (and all retraction) is avoided. These capabilities motivate a different kind of problem-solving architecture in which multiple potential solutions are explored simultaneously. This architecture is particularly well-suited for tasks where a reasonable fraction of the potential solutions must be explored.

1. Introduction

Most problem solvers search. If it were otherwise, a direct algorithm would solve the task. Problem solvers are constantly confronted with the necessity to select among equally plausible alternatives. These may concern the selection of which goal to try to achieve next, which plausible inference to draw from incomplete data, which hypothetical world to explore next, which action to perform, etc. Ultimately most, and often all, of these alternatives will be demonstrated to be incorrect. However, at the point the problem solver first encounters them, the alternatives appear equally likely. In this view, the problem solver is searching a space where each dimension is defined by a set of alternatives it has encountered. This view raises two questions to which the body of this paper is an answer: "How can this space be searched efficiently, or how can maximum information be transferred from one point in the space to another?" and "How, conceptually, should the problem solver be organized?"

One method for exploring the space, generate-and-test, simply enumerates every point in the search space and tests to see whether it satisfies the goal. This method, although extremely inefficient, is guaranteed to find a solution if

Artificial Intelligence **28** (1986) 127–162

there is one (for infinite spaces the enumeration must be appropriately ordered for this to be true). If each point of the search space is dissimilar this is the best that can be achieved. However, for most tasks there is a great deal of similarity among the points of the search space. As a consequence efficiency can be gained by carrying results obtained in one region of the search space into other regions. This is not an easy task. Unless the problem solver is carefully designed, the increased efficiency is achieved at the cost of coherency and exhaustivity.

Often the efficiency is gained by incorporating special case routines which detect similarities for the particular task domain at hand. The resulting problem solver is incoherent as the presence of the special case routines confuse content rules referring to the search space with control rules looking for similarities. Even more devastating, the complexities of the resulting system make it impossible to guarantee that a solution will be found in all the cases where one exists (i.e., exhaustivity is lost).

These concerns helped prompt the development of truth maintenance systems [12] (also called belief revision or reason maintenance systems). These systems forced a clean division within the problem solver between a component solely concerned with the rules of the domain and a component solely concerned with recording the current state of the search (i.e., the TMS). These TMSs allow designers to increase the efficiency of problem solvers without giving up coherency or exhaustivity. This paper presents a new *assumption-based* TMS (ATMS) which, for many tasks, is more efficient than previous TMSs and has a more coherent interface between the TMS and the problem solver without giving up exhaustivity. In addition, the assumption-based TMS avoids the usual restriction that only one point of the search space can be examined at a time.

This introduction concludes with a discussion of the basic intuitions underlying the ATMS. The following section analyzes search and the role a conventional TMS can play. In particular, chronological backtracking is contrasted with dependency-directed backtracking. Good as they are, all conventional TMSs suffer, to some extent, from shortcomings which, for many problem-solving tasks, are avoidable. The ATMS, which overcomes these shortcomings, is then presented in detail. The paper concludes with an overall discussion of the ATMS algorithms plus implementation details which significantly affect efficiency.

De Kleer shows in [7] how the basic ATMS framework supports a variety of styles of default reasoning. It also extends the basic ATMS to correctly deal with arbitrary propositional expressions. In [8], de Kleer presents a protocol for problem-solver-TMS interaction which assures that the advantages of the TMS are not accidentally lost. It includes an extensive comparison of the ATMS with other TMSs.

1.1. Basic results developed in this paper

Consider a conventional, *justification-based*, TMS such as presented in [12]. The basic architectural presupposition is that the overall reasoning system consists of two components: A problem solver which draws inferences and a TMS which records these inferences (called justifications). The TMS serves three roles in this overall system. First, it functions as a cache for all the inferences ever made. Thus inferences, once made, need not be repeated, and contradictions, once discovered, are avoided in the future. Second, the TMS allows the problem solver to make nonmonotonic inferences (e.g., "Unless there is evidence to the contrary infer A"). The presence of nonmonotonic justifications requires that the TMS use a constraint satisfaction procedure (called truth maintenance) to determine what data is to be believed. Third, the TMS ensures that the database is contradiction-free. Contradictions are removed by identifying absent justification(s) whose addition to the database would remove the contradiction. The added justifications are attached to the antecedents of the nonmonotonic justifications. The procedure of identifying and adding justifications to remove contradictions is called dependency-directed backtracking. The ATMS extends the cache idea, simplifies truth maintenance and avoids most dependency-directed backtracking. The ATMS can be viewed as an intelligent cache, or a very primitive learning scheme.

In a justification-based TMS a status of *in* (believed) or *out* (not believed)[1] is associated with every problem-solver datum. The entire set of in data define the current context. Therefore, the contexts under which a particular datum is believed are implicit. This requires that the database always be kept consistent, makes it impossible to refer to problem-solving contexts explicitly and requires truth maintenance and dependency-directed backtracking to move to a different point in the search space.

On the other hand, in an ATMS each datum is labeled with the sets of assumptions (representing the contexts) under which it holds. These assumption sets are computed by the ATMS from the problem-solver-supplied justifications. The idea is that the assumptions are the primitive data from which all other data are derived. These assumption sets can be manipulated far more conveniently than the datum sets they represent. There is no necessity that the overall database be consistent; it is easy to refer to contexts and moving to a different point in the search space requires very little work.

The actions of the ATMS are best understood in terms of a spatial metaphor. The assumptions are the dimensions of the search space. A solution corresponds to a point in the space and a context, characterized by a set of assumptions, depending on its generality, defines a point, line, plane, volume,

[1] It is important to distinguish lack of belief in a datum from belief in the negation of a datum.

etc., in the space. When a derivation is made, the ATMS records it in the most general way so that it covers as large a region of the space as possible. Conversely when a contradiction is recorded the ATMS finds its most general form in order to rule out as much of the search space as possible. Much of the work of the ATMS is determining the most general and parsimonious ways to record data and contradictions. All this is to facilitate what must happen when the problem solver shifts contexts. The problem is: how much of what was believed in previous contexts can be believed in the new one. This is yet another instance of the frame problem. The new context includes all the derivations of the preceding contexts which depend solely on the assumptions of the current context. Determining the next context and its contents is expensive in a justification-based TMS. Context membership is a simple subset test in the ATMS.

We can carry over many of the data to the new context, but where should the problem solver start working? This is not a simple task because the next context is only partially filled out and we want to start the problem solver precisely at this boundary. Certainly if there are pending problem-solving inferences which remain applicable in the new context these should be done. However, this is still not sufficient because there may have been rules which were attempted and failed or only partially attempted in old contexts. In the current context these rules may now succeed or run further. If we ignored these rules, then the problem solver would miss solutions. If we were to rerun these rules, then we would make redundant inferences. It is difficult to identify rules that failed in this way, so most problem solvers using TMSs usually take the conservative approach and rerun rules.

This problem is not intrinsic to the TMS but rather to the boundary between the TMS and the problem solver. De Kleer, in [8], outlines an interface protocol which solves the preceding problem. The difficulties arise when the problem solver is permitted to make a control decision based on the statuses, assumptions and justifications of nodes *without informing the TMS.* In this new interface the problem solver may only refer to the *content* of the datum, and all control decisions must be communicated explicitly to the TMS. It is important to note that this protocol is not intended to restrict the capabilities of the problem solver, but rather to move the boundary between the problem solver and the TMS to a more natural place in which each need only be concerned with issues relevant to it. The result is a slightly more complex TMS and a far simpler problem solver. Nevertheless, this protocol could be applied to any TMS—it does not intrinsically depend on an assumption-based TMS.

This protocol views problem solving as a continuous process of compiling rules. The problem solver compiles the rules and the TMS runs them to determine the solutions. A rule examined by the problem solver, i.e. compiled, need *never* be examined again on the same data because it is already compiled within the TMS as a justification.

The ATMS and its interface protocol share a great deal of intellectual territory with RUP [14–16]. Our views of what an assumption is, what a solution is, and what a rule language should be like are very similar. However, RUP is oriented to problem solving using a single consistent context which is constantly changed and updated as problem solving progresses. The ATMS, on the other hand, is oriented towards problem solving in multiple contexts simultaneously. This is efficiently achieved by labeling each datum with the assumptions upon which it ultimately depends. This idea and its ramifications radically alters the conception and technology of problem solving.

1.2. Reasoning with inconsistency

Conventional TMSs insist on consistency: the subset of the database currently believed must be consistent. There are two basic reasons for this. The first, originating from procedurally encoding knowledge, insists that the problem solver has a single controlled focus. The second, originating from classical logic, insists that the presence of inconsistency renders all deductions meaningless. Both of these reasons can be defeated, thereby retracting the insistence on consistency.

A consistent database ensures that the problem solver is working on only one part of the search space at a time. Consistency is a poor method for achieving control. The task of selecting the part of the search space to examine is the problem solver's, the TMS should only record the state of the search so far.

In classical logic, the presence of an inconsistency allows one to prove (and disprove) all assertions. However, few problem solvers using a TMS would deduce all things if a contradiction were allowed to persist. Assertions derived from data unaffected by the contradiction remain useful. Only assertions directly affected by the contradiction should be retracted. The fundamental difficulty is that it is costly to determine which data are affected. Thus the insistence on a single consistent state. In an assumption-based TMS, one can tell directly whether an assertion is affected or not, so demanding consistency is unnecessary.

This is best illustrated by an example. Suppose we had the theory consisting of:

$$a, \qquad c,$$

$$a \rightarrow b, \qquad c \rightarrow d,$$

$$a \wedge c \rightarrow e, \qquad b \wedge d \rightarrow \bot.$$

$\bot$, $\rightarrow$, and $\wedge$ designate falsity, material implication, and conjunction. This theory is inconsistent as it allows the derivation of $\bot$. As there is no way to remove the contradiction, in a sense, everything is affected by the contradiction. Suppose the example were reformulated:

$$:MA/A\,, \qquad :MC/C\,,$$

$$A \rightarrow b\,, \qquad C \rightarrow d\,, \qquad A \wedge C \rightarrow e\,,$$

where $:M$ is Reiter's [18] default operator. $:MA/A$ indicates A (by convention capital letters represent defaults) is believed unless there is evidence to the contrary. This default theory has exactly one *extension* containing the atoms:

$$\{A,b,C,d,e\}\,.$$

The addition of the final axiom has a very different effect than in the preceding formulation.

$$b \wedge d \rightarrow \bot\,.$$

This introduces a contradiction, but this contradiction only directly affects e, not anything else. Both defaults cannot hold simultaneously. The theory now has two extensions containing the atoms:

$$\{A,b\}\,, \qquad \{C,d\}.$$

Intuitively, the addition of the final axiom forced the extension to split into two.

The ATMS operates as suggested by this example. Each datum is labeled with the consistent sets of defaults (represented by ATMS assumptions) actually used to derive it:

$$\langle b,\{\{A\}\},\rangle\,, \qquad \langle d,\{\{C\}\},\rangle\,, \qquad \langle e,\{\ \},\rangle\,.$$

Before adding the final axiom, e was represented:

$$\langle e,\{\{A,C\}\},\rangle\,.$$

A contradiction merely indicates that a specific set of assumptions is inconsistent, and that data depending on them are affected. In this sense the ATMS is similar to systems of Martins [13] and Williams [22].

1.3. The history of the ATMS idea

The ATMS was born out of frustration. Qualitative reasoning and constraint languages, the primary topics of my research, both involve choosing among alternatives. Various packages for dealing with alternatives have been proposed, primarily derived from [12]. These systems have proven to be inadequate for even simple qualitative reasoning tasks. The reasons for this are

twofold. First, they are intrinsically incapable of working with multiple contradictory assumptions at once—something one needs to do all the time in qualitative reasoning. Second, they are inefficient.

LOCAL [3] is a program for troubleshooting electronic circuits which was incorporated in SOPHIE III [1]. It uses propagation of constraints to make predictions about device behavior from component models and circuit measurements. As the circuit is faulted, some component is not operating as intended. Thus, at some point, as the correct component model does not describe the actual faulty component, the predictions will become inconsistent. The assumptions are that individual components are functioning correctly. A contradiction implies that some assumption is violated, hence the fault is localized to a particular component set. The best measurement to make next is the one that provides maximal information about the validity of the yet unverified assumptions, This program requires that the assumptions of an inference be explicitly available and that multiple contradictory propagations be simultaneously present in the database. Hence, for this task the assumption-based approach is better.

QUAL [4, 6] and ENVISION [9] produce causal accounts for device behavior. QUAL can determine the function of a circuit solely from its schematic. Qualitative analysis is inherently ambiguous, and thus multiple solutions are produced. However, for any particular situation a device has only one function. QUAL selects the correct one by explicitly comparing different solutions—something that is only possible using assumption-based schemes.

By clarifying the ATMS ideas as reported in this paper, it is now possible to reexamine this earlier research. For example, [11] presents a new trouble-shooter based on the ATMS which handles multiple faults.

2. Search

This section discusses the various approaches for controlling search leading to the idea of a conventional TMS. Many of these examples are artificial and pathological. I have tried to reduce real problem-solving issues to the simplest possible examples so that the difficulties can be seen in their clearest form. De Kleer [5] presents these same issues in the abstract.

2.1. A simple constraint language

In order to explicate the ideas I define a simple constraint language to be used as a source of convenient examples. Each variable may have one of a fixed number of values, e.g.,

$$y \in \{-1,0,1\}.$$

Constraints utilize variables (single lower-case letters), integers, $=$, $\neq$, $+$, $-$, $\times$

(usually expressed as juxtaposition), and ! operators, e.g.,

$$x + y = z .$$

Inclusive disjunction is indicated by $\vee$, e.g.,

$$(x + y = z) \vee (z = 3) .$$

A very useful specialization of inclusive disjunction adds the stipulation that no two of the disjuncts hold simultaneously. I term this *oneof* disjunction as it is equivalent to specifying a disjunction in which exactly one of the disjuncts hold. In the binary case, oneof disjunction is equivalent to conventional exclusive disjunction. Oneof disjunction is notated by $\oplus$, e.g.,

$$(x + y = z) \oplus (z = 3) .$$

Note that,

$$x \in \{-1,0,1\} ,$$

is thus equivalent to,

$$(x = -1) \oplus (x = 0) \oplus (x = 1) .$$

The functions $e_i(x)$ require expensive computation: $e_i(x) = (x + 100000)!$. The goal of the problem solving is to discover all assignments of values to variables that satisfy the constraints. Values are always rational numbers, and usually integers.

2.2. Approaches to efficient search

Consider the following (admittedly pathological problem):

(1) $x \in \{0,1\}$, (2) $a = e_1(x)$,

(3) $y \in \{0,1\}$, (4) $b = e_2(y)$,

(5) $z \in \{0,1\}$, (6) $c = e_3(z)$,

(7) $b \neq c$, (8) $a \neq b$.

The simplest and most often used search strategy is exponential: Enumerate all the possibilities and try each one until a solution is found (or all solutions are found). In the example, there are 3 binary selections giving $2^3 = 8$ tentative solutions to test ('$*$' indicates a valid solution and '$\bullet$' indicates a contradiction):

$x = 0, \quad y = 0, \quad z = 0, \ldots \bullet$

$x = 0, \quad y = 0, \quad z = 1, \ldots \bullet$

$x = 0, \quad y = 1, \quad z = 0, \ldots *$

$x = 0, \quad y = 1, \quad z = 1, \ldots \bullet$

$x = 1, \quad y = 0, \quad z = 0, \ldots \bullet$

$x = 1, \quad y = 0, \quad z = 1, \ldots *$

$x = 1, \quad y = 1, \quad z = 0, \ldots \bullet$

$x = 1, \quad y = 1, \quad z = 1, \ldots \bullet$

As each solution requires 3 expensive computations, 24 expensive computations are required total.

Chronological backtracking, which only requires the additional machinery of a stack of variable bindings, improves efficiency. This depth-first problem solver processes the expressions in the order presented. If the expression is a selection, then try the first one, otherwise evaluate the equation. If the expression is inconsistent, back up to the most recent selection with a remaining alternative and resume processing expressions from that point. In this example, the search space of assumptions is:

(1) $x = 0$

(2) $x = 0, \quad y = 0$

(3) $x = 0, \quad y = 0, \quad z = 0 \bullet$

(4) $x = 0, \quad y = 0, \quad z = 1 \bullet$

(5) $x = 0, \quad y = 1$

(6) $x = 0, \quad y = 1, \quad z = 0 *$

(7) $x = 0, \quad y = 1, \quad z = 1 \bullet$

(8) $x = 1$

(9) $x = 1, \quad y = 0$

(10) $x = 1, \quad y = 0, \quad z = 0 \bullet$

(11) $x = 1, \quad y = 0, \quad z = 1 \bullet$

(12) $x = 1, \quad y = 1$

(13) $x = 1, \quad y = 1, \quad z = 0 \bullet$

(14) $x = 1, \quad y = 1, \quad z = 1 \bullet$

The extra machinery (a stack) for controlling the search is well worth it. Only 14 expensive computations are required, while the brute-force technique requires 24. One early application of chronological backtracking was QA4 [19]. It is the central control mechanism of PROLOG [2].

Examining the actions of the problem solver we observe that eight of the expensive computations are easily avoided[2]:

Futile backtracking. Steps (4) and (14) are futile. The selection $z \in \{0,1\}$ has no effect on the contradiction in steps (3) or (13), so steps (4) and (14) could have been safely ignored. When a contradiction is discovered the search should backtrack to an assumption which contributed to the contradiction, not to the most recent assumption made.

Rediscovering contradictions. Steps (10) and (14) are futile. When step (3) contradicted, it could have been determined that the contradiction only depended on $y = 0$ and $z = 0$, and that x's value was irrelevant. Thus step (10), which has the same values for y and z should never have been attempted. By a similar argument, step (14) is shown to be futile.

Rediscovering inferences. The factorial computations of steps (6), (7), and (9)–(14) are unnecessary. For example, step (6) repeats the factorial computation evoked by $z = 1$ of step (4). Backtracking erased the earlier factorial, but if the previous results could somehow be cached each factorial would only need to be computed once.

Incorrect ordering. Steps (3), (4), (13), and (14) are futile. If the test $a \neq b$ were moved just after the computation of b, 4 expensive computations would be eliminated.

These four problems are not independent. For example, carefully ordering the equations eliminates step (3), but step (3) computed a value for c which could have been used to eliminate step (6). In this example, careful ordering avoids futile backtracking, but this is not true in general.

TMSs, by exploiting dependencies, completely solve the first three defects and partially resolve the ordering problems. In the preceding example, solving the first three defects leaves only one futile step, (3), due to incorrect ordering. The scheduling strategy proposed in [8] for the problem-solver-TMS eliminates step (3) as well.

A solution to the defects is enabled by maintaining records of the dependence of each inference on earlier ones. When a contradiction is encountered these dependency records are consulted to determine which selection to backtrack to. Consider step (3). The dependency records indicate that $x = 0$ and $y = 0$ contribute to the contradiction, but $z = 0$ does not. Then the

[2] Mathematical analysis of the equations would reduce this set even further. Each e_i is the same function, so the number of expensive computations could be reduced to two. Furthermore, $e_i(x)$ is a one-to-one function, thus the solutions can be determined for x, y, z without performing a single expensive computation.

problem solver can backtrack to the most recent selection which actually contributed the contradiction. This technique is called dependency-directed backtracking [20].

The assumption set (e.g., $\{x = 0, y = 0\}$) which caused the contradiction is recorded so that the set can be avoided in the future. These are called the *nogood* sets [21] as they represent assumptions which are mutually contradictory.

Dependency records are bidirectional, linking antecedents to consequents as well as consequents to antecedents, thus the problem of rediscovering inferences is also avoided. Thus, if $x = 1$ is believed, the previously derived $a =$ 'A big number' is as well. Whenever some assumption is included in the current set, the dependency records are consulted to reinstate the previously derived consequents of that assumption. This can be effected quite directly within the database. Entries are marked as temporarily unavailable (i.e., out) if they are not derivable from the set of assumptions currently being explored. In addition, each new assumption set is checked to see whether it contains some known contradiction (i.e., has a subset in the nogood database).

These techniques are the basis of all the TMSs. In the more general TMS strategies it is not necessary to specify the overall ordering of the search space (so far we have been presuming some simple-minded enumeration algorithm). They, in effect, choose their own enumeration but this ordering can be controlled somewhat by specifying which parts of the search space to explore first.

It is important to note that all these strategies are ultimately equivalent. The most sophisticated TMS will find as many consistent solutions as pure enumeration. The goal is to enhance efficiency without sacrificing exhaustivity.

3. Limitations of Current TMSs

Truth maintenance systems are the best general-purpose mechanisms for dealing with assumptions. However, when designing a TMS mechanism one must be careful to avoid certain pitfalls. All previous TMS mechanisms which are known to the author suffer from one or more of the limitations listed in Section 3.2. However, the ATMS presented in this paper provides a simultaneous solution to all of these problems. De Kleer, in [8], analyzes the systems of Doyle [12], Martins [13], McAllester [15], McDermott [17], and Williams [22] and points out how these TMSs have addressed or failed to address these problems. He also discusses in [8] some of the costs associated with using the ATMS. The ATMS is not a panacea and is not suited to all tasks. Conventional TMSs are oriented to finding one solution, and extra cost is incurred to control the TMS to find many solutions. The ATMS is oriented to finding all solutions, and extra cost is incurred to control the ATMS to find fewer solutions.

3.1. Doyle's TMS

This paper adopts much of Doyle's nomenclature, so this section presents a thumbnail (and somewhat simplistic) sketch of his TMS.

The TMS associates a special data structure, called a node, with each problem-solver datum (which includes database entries, inference rules, and procedures). Although the node is a complex data structure from the TMS' point of view, as far as the problem solver using the TMS is concerned, it is just a convenient way of referring to the belief in a datum. Justifications represent inference steps from combinations of nodes to another node. The simplest kind of justification, an SL-justification, consists of an inlist and an outlist. A node is believed, or in, if it has a valid justification. A justification is valid, if each of the nodes of its inlist are in and each of the nodes of its outlist are out. The problem solver can add nodes, justifications and mark nodes as contradictory at any time.

The TMS provides two services: truth maintenance and dependency-directed backtracking. Truth maintenance finds an assignment of belief statuses (i.e., in or out) for every node such that every justification is satisfied. However, each node must have well-founded support, so circular supports which may satisfy the justifications locally are outlawed. Although a node may have many justifications, only a few may hold, and one of these is chosen as the current supporting justification. A contradiction is encountered when a node, marked as contradictory, is assigned belief. Dependency-directed backtracking searches for the assumptions contributing to the contradiction. An assumption is a node with a current supporting justification with a nonempty outlist. The contradiction is removed by picking one assumption, the culprit, and adding a justification for one of the nodes of its outlist by a conjunction of the culprit's inlist and the remaining assumptions (see [8] for more details).

3.2. TMS limitations

The single-state problem. Given a set of assumptions which admits multiple solutions, the TMS algorithms only allow one solution to be considered at a time. This makes it extremely difficult to compare two equally plausible solutions. For example, $\{x = 0, y = 1, z = 0\}$ and $\{x = 1, y = 0, z = 1\}$ are both solutions to the preceding problem. It is impossible to examine both of these solutions together. However, this is often exactly what one wants to do in problem solving—differential diagnosis to determine the best solution.

Overzealous contradiction avoidance. Suppose A and B are contradictory. The TMS will guarantee that either A or B will be worked on, but not both. This is not necessarily the best problem-solving tactic. All a contradiction

between A and B indicates is that inferences dependent on both A and B be avoided. But it is still important to draw inferences from A and B independently.

Switching states is difficult. Suppose that the problem solver decides to temporarily change an assumption (i.e., not in response to a contradiction). There is no efficient mechanism to facilitate this. The only direct way to change assumptions is to introduce a contradiction, but once added it cannot be removed so the knowledge state of the problem solver is irreconcilably altered. Suppose a change out of the current state was somehow achieved; there is no way to specify the target state. All a TMS can guarantee is that the database state is contradiction-free. So, in particular, there is no way to go back to a previous state. The reason for these oddities is that a TMS has no useful notion of global state. One inelegant mechanism that is sometimes utilized to manipulate states is to take snapshots of the status and justifications of each datum then later reset the entire database from the snapshot. This approach is antithetical to the spirit of TMS for it reintroduces chronological backtracking. Information garnered within one snapshot is not readily transferred to another.

The dominance of justifications. What Doyle [12] calls an assumption is context-dependent: an assumption is any node whose current supporting justification depends on some other node being out. As problem solving proceeds, the underlying support justifications and hence the set of nodes considered assumptions changes. This is problematic for problem solvers which frequently consult the assumptions and justifications for data.

The machinery is cumbersome. The TMS algorithm can spend a surprising amount of resources finding a solution that satisfies all the justifications. Determining the well-founded support for a node requires a global constraint satisfaction process. Detecting loops of odd numbers of nonmonotonic justifications (which provoke infinite computation loops) is expensive. The dependency-directed backtracking in response to a contradiction may require extensive search, and the resolution of the contradiction often results in other contradictions. Eventually, all contradictions are resolved, but only after much backtracking. During this time the status of some datum may have changed between in (believed) and out (not believed) many times.

Unouting. As problem solving progresses a datum can be derived, retracted when a contradiction occurs, and then reasserted when a second contradiction causes the current context to switch again (a common occurrence in problem solving). The process of determining which retracted, previously derived, data can be reasserted is called unouting. The maintenance of dependency records avoids having to rediscover these inferences. Unfortunately, unless great care is taken at the problem-solver-TMS interface, some previously discovered data will be rederived.

Consider the following simple (again pathological) example:

(1) $x \in \{1,2\}$

(2) $y \in \{1,2)$

(3) $x \neq y \oplus x = y$

(4) $z = (1000xy)!$

(5) $r = (100xy)!$

Lines (1), (2), and (3) each represent a binary selection. The problem solver starts with the first assumption[3] of each selection: $\{\Gamma_{x=1}, \Gamma_{y=1}, \Gamma_{x\neq y}\}$. The first two equations immediately assign values to x and y leaving the three equations $x \neq y$, $z = (1000xy)!$, and $r = (100xy)!$. None of these equations depends on the result of a previous one, thus each of them can be done first. The problem solver could first compute $z = 1000!$, then $r = 100!$, and finally notice that $x \neq y$ provoked a contradiction. The computation of $z = 1000!$ is recorded with a dependency record indicating that it depends on $\{\Gamma_{x=1}, \Gamma_{y=1}\}$. Later when the assumption set $\{\Gamma_{x=1}, \Gamma_{y=1}, \Gamma_{x=y}\}$ is explored, these dependency records are consulted and $z = 1000!$ is reasserted without having to recompute it.

The difficult case occurs if the original order were, instead, $z = (1000xy)!$, $x \neq y$, and $r = (100xy)!$. As a result $z = 1000!$ is computed, the contradiction is detected, but $r = 100!$ is *not* computed. Sometime later the assumption set $\{\Gamma_{x=1}, \Gamma_{y=1}, \Gamma_{x=y}\}$ is examined. The previously derived $z = 1000!$ is unouted, but $r = 100!$ is not because it was never derived. We have a dilemma: If we do nothing $r = 100!$ is not computed, but if we restart the problem solver on $\{\Gamma_{x=1}, \Gamma_{y=1}, \Gamma_{x=y}\}$ we derive $z = 1000!$ twice. This is the unouting problem. Put abstractly, if the same datum is reexamined in a new context, it is insufficient to just reexamine its previous consequents to see if they hold in the new context because there may be consequents not derivable in the old context, which are derivable in the new. The difficult task is how to fill the "gaps" of the consequents without redoing the entire computation.

There have been four styles of solutions to this task, none completely satisfactory.

(1) Even though a contradiction occurs during analysis of $\{\Gamma_{x=1}, \Gamma_{y=1}, \Gamma_{x\neq y}\}$ this computation could be allowed to go on. The difficulty here is that a great deal of effort may be spent working on a set of assumptions that may be irrelevant to an overall solution.

(2) Another technique is to store a snapshot of the problem solver's state (its pending task queue) which can be reactivated at a later time.

(3) It is also possible to restart the computation from $\{\Gamma_{x=1}, \Gamma_{y=1}\}$, taking advantage of the previous result by examining existing consequents to determine ones are missing.

[3] For perspicuity, when there is no ambiguity, Γ_n is used to represent the assumption n.

(4) The easiest technique is just to restart the computation from $\{\Gamma_{x=1}, \Gamma_{y=1}\}$ without taking any effort to see which consequents should be unouted. All expensive problem-solving steps are memoized so no time-consuming steps are repeated. For example, de Kleer and Sussman, [10], use this technique to cache all symbolic GCD computations.

None of these four solutions is completely general or optimal. De Kleer, [8], presents a problem-solver-TMS protocol which, if followed, allows the TMS to control the problem solver so that no inference is ever done twice.

4. The Basic ATMS

Most of the preceding TMS problems result from the inability to refer directly to the contexts in which a datum holds. As a consequence the database must be kept contradiction-free, describe one context at a time, and cannot be labeled so that it can be reexamined later. The fundamental reason for this is that a datum is solely marked with a set of justifications which specify how it derives from other data, but which only implicitly describe the contexts in which it holds. The solution proposed here augments the justifications of a conventional TMS with a label which explicitly describes the contexts under which the datum holds. This label is updated and manipulated along with the justifications.

4.1. Problem-solver architecture

The ATMS adopts the same view of problem solving as a conventional TMS. The reasoning system (or *overall* problem solver) consists of two components: a problem solver and a TMS (Fig. 1). The problem solver is usually at least first order and includes all domain knowledge and inference procedures. Every inference made is communicated to the TMS. The TMS' job is to determine what data are believed and disbelieved given the justifications recorded thus far. The TMS performs this task through propositional inference using the justifications as propositional axioms.

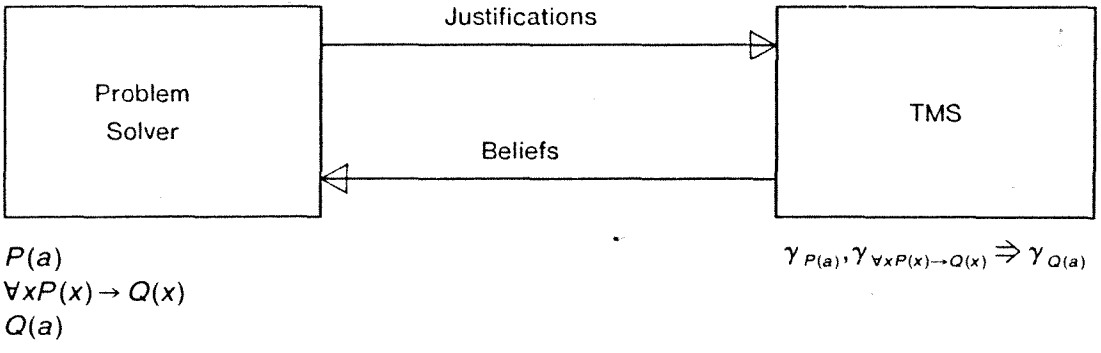

FIG. 1. Reasoning system.

Fig. 1 highlights two properties which are crucial to understanding the use of TMSs in reasoning systems. First, the same expressions have internal structure to the problem solver but are atomic to the TMS. For example, the inference $Q(a)$ is made by matching the structure of $P(a)$ with $\forall x P(x) \rightarrow Q(x)$. When this inference is recorded as a justification, the TMS treats the expressions as uninterpreted symbols or propositional atoms. The TMS task is solely to determine what symbols follow from what given the propositional axioms (the justifications). Thus there are two inference procedures in the reasoning system both operating on the same expressions but treating them entirely differently. Second, problem solving is a process of accumulating justifications and changing beliefs until some goal is satisfied. The TMS determines belief *given the justifications encountered thus far* and *not with respect to the logic of the problem solver*. The addition of a justification may cause any belief to change. Problem solving is a continual process of consulting the current beliefs to decide what inference to make next, recording the inference as a justification, and redetermining the current set of beliefs. The task of the problem-solver designer is to ensure that this process converges to a solution which satisfies the problem-solving goals.

4.2. Basic definitions

Crucial to the ATMS architecture is a subtle distinction that, to this point in the paper, has been left implicit. It is critical to distinguish the thing assumed from the decision to assume. Thus far, the word "assumption" has been used to designate both of these concepts. An *assumption* is restricted to designate a decision to assume without any commitment as to what is assumed. An *assumed* datum is the problem-solver datum that has been assumed to hold. Assumptions are connected to assumed data via justifications as discussed in the next section.

Assumptions form the foundation to which every datum's support can be ultimately traced. Intuitively, it thus makes little sense to allow assumptions themselves to be derived. This point of view has provoked considerable controversy. Therefore, the ATMS permits assumptions to be derived if need be, although this paper avoids utilizing this feature.

If an assumption is uniquely associated with a particular datum x, then, for purposes of perspicuity, this assumption is written Γ_x. Not all assumptions can be so represented, the same assumption may justify multiple data or one datum may be justified by multiple assumptions.

An ATMS *node* corresponds to a problem-solver datum. An assumption is a special kind of node.

An ATMS *justification* describes how a node is derivable from other nodes. A justification has three parts: the node being justified, called the *consequent*; a list of nodes, called the *antecedents*, and the problem solver's

description of the justification, called the *informant* (its utility is illustrated in [8]). Justifications are written as:

$$x_1, x_2, \ldots \Rightarrow n .$$

Here $x_1, x_2, \ldots$ are the antecedent nodes and n is the consequent node. A justification is a material implication,

$$x_1 \wedge x_2 \wedge \cdots \rightarrow n,$$

where $x_1, x_2, \ldots, n$ are atoms. Thus, the justifications are propositional Horn clauses. The nonlogical notation "$\Rightarrow$" is used because the ATMS does not allow negated literals and treats implication unconventionally.

An ATMS *environment* is a set of assumptions. Logically, an environment is a conjunction of assumptions.

A node n is said to hold in environment E if n can be derived from E and the current set of justifications J. Derivability is defined in terms of the usual propositional calculus,

$$E,J \vdash n ,$$

where E is viewed as a conjunction of atoms and J is viewed as a set of implications.

An environment is inconsistent if false (notated $\perp$) is derivable propositionally:

$$E,J \vdash \perp .$$

An ATMS *context* is the set formed by the assumptions of a consistent environment combined with all nodes derivable from those assumptions.

A *characterizing environment* for a context is a set of assumptions from which every node of the context can be derived. Every context has at least one characterizing environment. Usually (when assumptions have no justifications) every context has exactly one characterizing environment.

Given these basic definitions, the goals of the ATMS can be conceptually outlined. The ATMS is provided with a set of assumptions and justifications. The task of the ATMS is to efficiently determine the contexts. Efficiency is achieved by taking advantage of two important observations about TMS use. First, the problem solver is supplying the ATMS with justifications and assumptions one at a time. Therefore, the ATMS is incremental, updating only the changed contexts. Second, it is extremely rare for a problem solver to ask for the contents of a context. Problem solvers usually only require to be able to tell when a context becomes inconsistent and whether a node holds in a

particular context. Therefore, instead of explicitly determining the contexts, the ATMS constructs an intermediate data structure which makes context-consistency checking and node inclusion very fast.

The ATMS constructs this data structure by associating descriptions of contexts with data, instead of the usual association of data with contexts. The ATMS associates with every datum a parsimonious description of every context in which the datum holds. This organization has the advantage that it is extremely convenient for the problem solver to work in all contexts at once (this is discussed further in [8]).

4.3. Labels

An ATMS *label* is a set of environments associated with every node. Every environment E of n's label is consistent and has the property that:

$$E,J \vdash n.$$

More intuitively,

$$J \vdash E \rightarrow n.$$

The label describes the assumptions the datum ultimately depends on and, unlike the justification, is constructed by the ATMS itself. While a justification describes how the datum is derived from immediately preceding antecedents, a label environment describes how the datum ultimately depends on assumptions. These sets of assumptions can be computed from the justifications, but computing them each time would destroy the efficiency advantages of this ATMS.

The task of the ATMS is to guarantee that each label of each node is consistent, sound, complete and minimal with respect to the justifications. A label is consistent if all of its environments are consistent. A label for node n is sound if n is derivable from each E of the label:

$$J \vdash E \rightarrow n.$$

The label for node n is complete if every consistent environment E for which,

$$J \vdash E \rightarrow n,$$

is a superset of some environment E' of n's label (viewing environments as sets):

$$E' \subset E.$$

The label is minimal if no environment of the label is a superset of any other. For any label environment E there should not exist any other label environment E' such that:

$$E' \subset E.$$

As a consequence of these definitions, node n is derivable from environment E in exactly those cases that E is a superset of any environment of the label. This property has an important consequence for the problem solver manipulating the data. It can directly tell whether a node holds in some environment or not. A node has an empty label iff it is not derivable from a consistent set of assumptions. A node has a nonempty label iff it is derivable from a consistent set of assumptions.

A node is a member of a context if it can be derived from the assumptions of an environment characterizing the context. The representation makes it easy to determine this: A node is in the context if it has at least one environment which is a subset of an environment characterizing the context. Thus a node implicitly specifies all the contexts it can be a member of: namely, those contexts having a characterizing environment which are a superset of one of its label's environments. One of the basic advantages of this ATMS is that membership within a context can be checked by a direct subset test which itself can be implemented efficiently.

Each context implicitly separates all the nodes into three categories: nodes necessarily present, nodes necessarily absent, and nodes currently absent but which may be become necessarily present or absent with subsequent justifications. A node is necessarily absent if its addition would cause the derivation of $\perp$. Assumptions necessarily absent are easily identified. If a nogood is a superset of a context's assumptions, then those assumptions of the nogood not mentioned by the context form a conjunction of assumptions which cannot consistently extend the context. It is harder to identify other necessarily absent nodes, however, if a necessarily absent assumption has only a single consequent, then that assumed datum is necessarily absent.

Consequently, the set of nodes can be separated into four nonoverlapping sets (membership is easily computable). The *true* set contains nodes which hold in the empty environment. These nodes must hold in every consistent context which has or will be found. This set grows monotonically. The *in* set contains nodes whose label has at least one nonempty environment. These nodes hold in at least one nonuniversal consistent context. However, as problem solving proceeds these contexts may be discovered to be inconsistent. Thus, this set grows nonmonotonically. The *out* set contains nodes whose label is empty. These nodes hold in no known consistent context. Subsequent problem solving may discover a new context for a node moving the node to the in set, or discover a context is inconsistent moving some in node to the out set. The *false* set corresponds to nodes

which do not hold in any context which has or will be found. This set grows monotonically.

The flow of nodes between these sets is straightforward. Every new node is created with an empty label and thus as a member of the out set. Problem solving may move out and in nodes to any other set, but no problem solving can remove nodes from the true or false set.

4.4. Basic data structures

The basic data structure is an ATMS node. It contains the problem-solver datum with which it is associated, the justifications the problem solver has created for it, and a label computed for it by the ATMS:

$$\gamma_{\text{datum}}\text{: } \langle \text{datum,label,justifications} \rangle .$$

γ_x designates the node with datum x. However, in cases where it makes little difference the same designation is used to represent both the node and its datum. The label is computed by the ATMS and must not be changed by the problem solver. The datum is supplied by the problem solver, and is never examined by the ATMS. The justifications are supplied by the problem solver and are examined but never modified by the ATMS.

All nodes are treated identically, and are distinguished only by their individual pattern of label environments and justifications. There are four types of nodes: *premises*, *assumptions*, *assumed nodes*, and *derived nodes*.

A premise has a justification with no antecedents, i.e., it holds universally. The node,

$$\langle p, \{\{\}\}, \{()\} \rangle$$

represents the premise p. Although new justifications can be added to a premise, this will not affect the label unless the empty environment becomes inconsistent.

An assumption is a node whose label contains a singleton environment mentioning itself. The node,

$$\langle A, \{\{A\}\}, \{(A)\} \rangle ,$$

represents the assumption A (lower-case letters and γ_x represent any node and upper-case letters and Γ_x represent assumptions). Assumptions may be justified:

$$\langle A, \{\{A\}, \{B, C\}\}, \{(A), (d)\} \rangle .$$

An assumed node is neither a premise nor an assumption and has a justification mentioning an assumption. The assumed datum a which holds under assumption A is represented,

$$\langle a,\{\{A\}\},\{(A)\}\rangle .$$

Preferably, defeasible problem-solving nodes should be created by assumption-assumed-node pairs. This technique avoids having to ever justify assumptions and makes it convenient to remove unnecessary assumptions if need be (i.e., by contradicting the assumption, not the assumed node).

All other nodes are derived. The derived node,

$$\langle w = 1,\{\{A,B\},\{C\},\{E\}\},\{(b),(c,d)\}\rangle$$

represents the fact that $w = 1$ is derived from either the node b or the nodes c and d. In addition, the node holds in environments $\{A, B\}$ and $\{C\}$ and $\{E\}$.

Justifications can be added to any type of node, conceivably changing its type. However, purely for efficiency, as assumptions require a more elaborate data structure, nonassumptions cannot become assumptions, or assumptions cannot become another type of node. If that capability is needed, the assumption-assumed-node technique should be used.

Logically, the label represents a material implication. The label of

$$\langle n, \{\{A_1, A_2, \ldots\}, \{B_1, B_2, \ldots\}, \ldots\}, \{(z_1, z_2, \ldots)(y_1, y_2, \ldots)\ldots\}\rangle$$

represents the implication,

$$(A_1 \wedge A_2 \wedge \cdots) \vee (B_1 \wedge B_2 \wedge \cdots) \vee \cdots \rightarrow n .$$

Similarly, the justifications represent the implication,

$$(z_1 \wedge z_2 \wedge \cdots) \vee (y_1 \wedge y_2 \wedge \cdots) \vee \cdots \rightarrow n .$$

An out set node may have justifications if no justification holds in a consistent environment. Such a node must not be removed from the database because future ATMS processing (in response to new justifications) may produce a consistent environment for it.

The distinguished node,

$$\gamma_\perp : \langle \perp, \{\}, \{\ldots\}\rangle ,$$

represents falsity. The inconsistent environments are called *nogoods* because they represent inconsistent conjunctions of assumptions. Note that the nogoods

are the label $\gamma_\perp$ would have if inconsistent environments were not excluded from node labels.

4.5. The environment lattice

Every consistent environment characterizes a context. If there are n assumptions, then there are potentially 2^n contexts. There are $\binom{n}{k}$ environments having k assumptions. Fig. 2 illustrates the environment lattice for assumptions $\{A, B, C, D, E\}$. The nodes represent environments. The upward edges from an environment represent subset relationships with environments one larger in size. Conversely, the downward edges from an environment represent superset relationships with environments one smaller. All the supersets of an environment can be found by tracing upward through the lattice, and all the subsets of an environment can be found by tracing downwards through the lattice. Thus, all environments are subsets of the top-most node $\{A, B, C, D, E\}$, and all environments are supersets of the bottom-most node $\{\}$.

Any environment which allows the derivation of $\perp$ is nogood and removed from the lattice. On figures, the crossed-out nodes correspond to nogoods. If an environment is nogood, then all of its superset environments are nogood as well. The nogoods of Fig. 2 are the result of the single nogood $\{A, B, E\}$.

The ATMS associates every datum with its contexts. If a datum is in a context, then it is in every superset as well (the inconsistent supersets are ignored). For example, the circled nodes of Fig. 2 indicate all the contexts of $\gamma_{x+y=1}$. The label of a node is the set of greatest lower bounds of the circled nodes:

$$\gamma_{x+y=1}: \langle x + y = 1, \{\{A, B\}, \{B, C, D\}\}, \{\ldots\}\rangle .$$

The square nodes of Fig. 2 correspond the contexts of $\gamma_{x=1}$:

$$\gamma_{x=1}: \langle x = 1, \{\{A, C\}, \{D, E\}\}, \{\ldots\}\rangle .$$

Suppose nothing is known about y, and the problem solver infers $y = 0$ from $x + y = 1$ and $x = 1$:

$$\gamma_{x+y=1}, \gamma_{x=1} \Rightarrow \gamma_{y=0} .$$

The contexts of $\gamma_{y=0}$ are the intersection of the contexts of $\gamma_{x+y=1}$ and $\gamma_{x=1}$. The label for $\gamma_{y=0}$ is the set of greatest lower bounds of the intersection:

$$\gamma_{y=0}: \langle y = 0, \{\{A, B, C\}, \{B, C, D, E\}\}, \{(\gamma_{x+y=1}, \gamma_{x=1})\}\rangle .$$

4.6. Basic operations

The three basic ATMS actions are creating an ATMS node for a problem-solver datum, creating an assumption, and adding a justification to a node. The

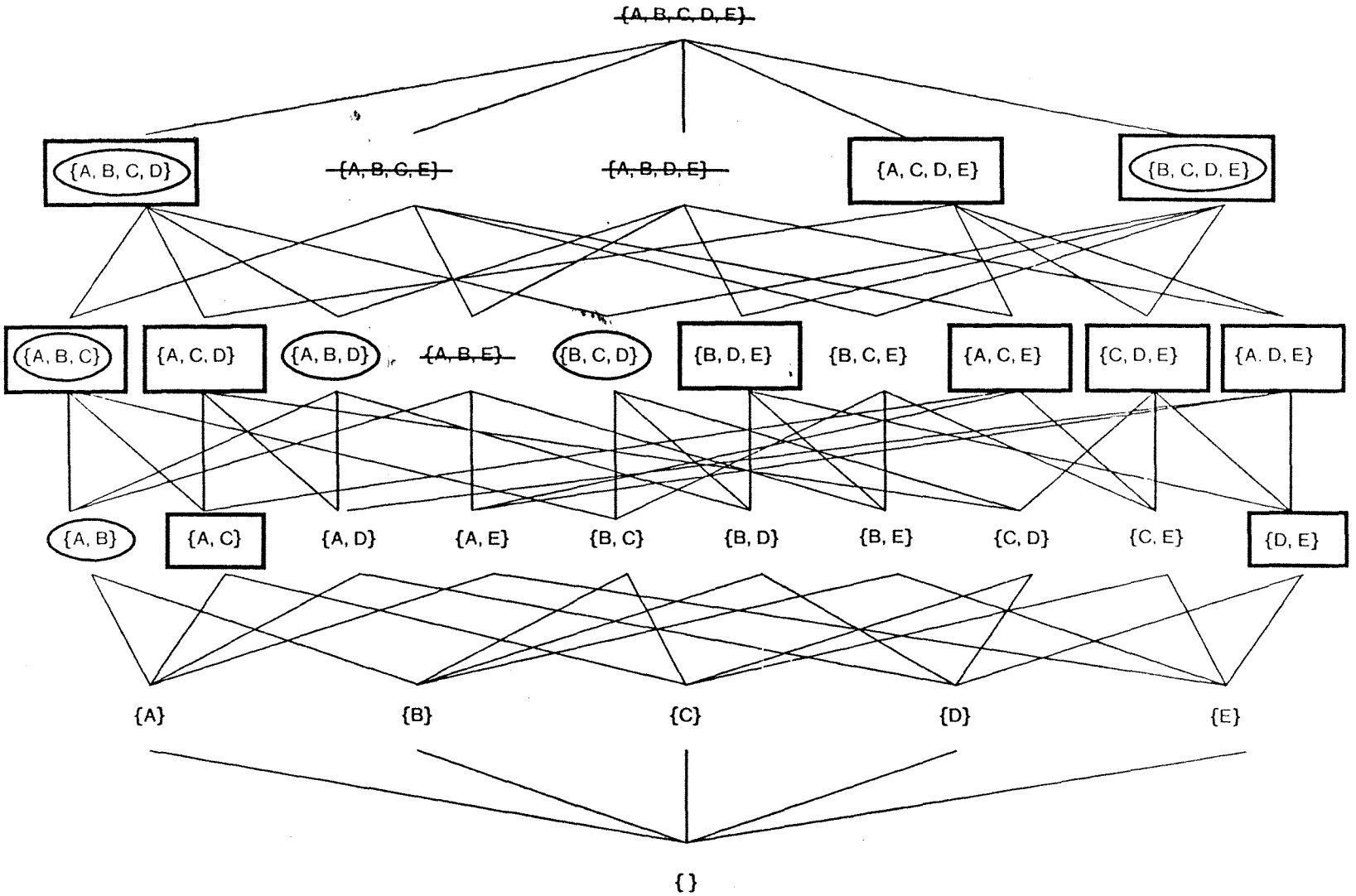

FIG. 2. Environment lattice.

ATMS is incremental. The algorithms ensure that, after every primitive operation, every label of every node is consistent, sound, complete and minimal. All ATMS operations can be built out of the three primitive operations.

Associating ATMS nodes with data is preliminary to any ATMS actions. It is crucial that two data which are the "same" be associated with the same ATMS node. This can only be achieved by some indexing scheme within the problem solver such that all forms of the same datum are associated with the same node. If this is not done, and each new derived datum is automatically associated with a new node, then many of the advantages of the ATMS disappear. Creating an assumption, like creating a node, requires little ATMS work. Identifying what assumptions should be created, however, can be a difficult task for the problem solver or its designer. The only primitive operation which can cause significant processing for the ATMS and the problem solver is adding a justification.

A node n is assumed by giving it a justification which includes an assumption, e.g.,

$$A \Rightarrow n .$$

Premises are represented as justifications with no antecedent nodes (hence their labels consist of the empty environment). The justification,

$$\Rightarrow p .$$

results in the node,

$$\langle p, \{\{\}\}, \{(), \ldots\}\rangle .$$

justifications for $\gamma_\perp$ indicate inconsistent conjunctions of nodes (i.e., $(n_1 \wedge n_2 \wedge \cdots \wedge n_k \rightarrow \gamma_\perp) \equiv \neg(n_1 \wedge n_2 \wedge \cdots \wedge n_k)$).

It is important to note that node labels are completely insensitive to the order in which the assumptions, nodes, and justifications are introduced.

4.7. Basic algorithms

The ATMS must ensure that for every justification the intersection of the contexts of the antecedents are equal to the contexts of the consequent. As the ATMS represents a context by its greatest lower bound, this computation is involved. Consider the example just discussed:

$$\gamma_{x+y=1}: \langle x + y = 1, \{\{A, B\}, \{B, C, D\}\}, \{\ldots\}\rangle ,$$

$$\gamma_{x=1}: \langle x = 1, \{\{A, C\}, \{D, E\}\}, \{\ldots\}\rangle ,$$

$$\gamma_{y=0}: \langle y = 0, \{\}, \{\}\rangle ,$$

$$nogood\{A, B, E\} .$$

(As nogoods are central to the ATMS algorithms, they are stored in a separate database.) As the problem solver deduces $y = 0$ from $x + y = 1$ and $x = 1$ it adds the justification:

$$\gamma_{x+y=1},\ \gamma_{x=1} \Rightarrow \gamma_{y=0}\ .$$

As a result of adding this justification, the ATMS updates $\gamma_{y=0}$'s label to be sound, complete, consistent, and in minimal form:

$$\gamma_{y=0}\colon \langle y = 0, \{\{A, B, C\}, \{B, C, D, E\}\}, \{(\gamma_{x+y=1}, \gamma_{x=1})\}\rangle\ .$$

One sound and complete label for the consequent is the set whose elements are the union of all possible combinations of picking one environment from each antecedent node label. Thus one sound and complete label for $\gamma_{y=0}$ is

$$\{\{A, B, C\}, \{A, B, C, D\}, \{A, B, D, E\}, \{B, C, D, E\}\}\ .$$

Any sound and complete label can be made consistent and minimal by removing subsumed and inconsistent environments. The environment $\{A, B, C, D\}$ is removed because it is subsumed by $\{A, B, C\}$. The environment $\{A, B, D, E\}$ is not included because it contains the inconsistent $\{A, B, E\}$.

In terms of contexts, the resulting consequent datum is not inserted into a current context, but rather in the most general context(s) (i.e. the greatest lower bounds of the intersection) in which it holds. Thus the consequent datum will carry over to many contexts other than one currently being worked upon.

The following is a description of a simple label-update algorithm which satisfies the ATMS goals. Processing starts with the problem solver supplying the ATMS with a new justification. First, a new label is computed for the node assuming every antecedent node has a sound and complete label. This label computation can be viewed as a set manipulation, or as a logical operation. In terms of sets, given j_{ik} the label of the ith node of the kth justification for consequent node n, a complete label for node n is:

$$\bigcup_k \left\{ x \mid x = \bigcup_i x_i \text{ where } x_i \in j_{ik} \right\}\ .$$

If the labels are viewed as propositional expressions in disjunctive normal form, a complete label is computed by converting the expression,

$$\bigvee_k \bigwedge_i j_{ik}\ ,$$

into disjunctive normal form. After removing inconsistent and subsumed environments, the label is sound and minimal. Second, if the newly computed label is the same as the old label, then we are finished with the node. Third, if

the node is $\gamma_\perp$, each environment of the label is added to a nogood database and all inconsistent environments (i.e., it and all its supersets) are removed from every node label. Fourth, if the node is not $\gamma_\perp$, then the updating process recurs updating the labels of all the consequent nodes (i.e., other nodes having justifications which mention the node whose label changed). Although this simple process may update the same node label more than once, the process must terminate because there are a finite number of assumptions. Note that labels are not guaranteed to be consistent and complete until the algorithm terminates.

There are many optimizations to this algorithm. The LISP implementation of this propagator takes more advantage of the fact that node labels are sound, consistent, complete and minimal to start with. It is not necessary to recompute the entire node label for each node update. It is sufficient to compute the incremental change in the label contributed by the new justifications. If a node has only one justification which mentions only one node with a nonempty label, then the label can just be copied forward.

In the design of the basic algorithm, each node has five slots:
- Datum: The problem solvers' representation of the fact.
- Label: A set of environments.
- Justifications: A list of the derivations of the datum.
- Consequents: A list of justifications in which the node is an antecedent.
- Contradictory: A single bit indicating whether the datum is contradictory.

In addition an environment has three slots:
- Assumptions: The set of assumptions defining it.
- Nodes: The set of nodes whose label mentions it.
- Contradictory: A single bit indicating whether the environment is contradictory.

A justification has three slots:
- Informant: A problem-solver-supplied description of the inference.
- Consequent: The node justified.
- Antecedents: A list of nodes upon which the inference depends.

4.8. Complexity considerations

The architecture of the ATMS is such that it is practical to use even when n is very large (e.g., 1000). The ATMS is then exploring a space of size 2^{1000}. To clarify how the ATMS can efficiently explore such large spaces consider the environment lattice. The efficiency of the ATMS arises from two important observations about this lattice. First, due to the fact that a node is in every superset context as well, it is only necessary to record the greatest lower bound environments in which a node holds (the label). Similarly, it is only necessary to record the greatest lower bounds of the inconsistent environments. Second, in cases

where most of the environments are inconsistent, these inconsistencies are identifiable in small subsets of assumptions. Therefore, large parts of the environment lattice need never be explicitly checked for consistency.

The efficiency of the ATMS is thus directly proportional to the number of environments it is forced to consider (thus actually constructing the environment lattice is always a bad idea). Ultimately, the observed efficiency of the ATMS is a result of the fact that it is not that easy to create a problem which forces the TMS to consider all 2^n environments without either doing work of order 2^n to set up the problem or creating a problem with 2^n solutions. To construct such a pathological case, one has to cleverly create a set of justifications such that every possible environment is a minimal nogood or a minimal environment for some node at some point in the process. De Kleer, in [8], considers these issues in more detail.

4.9. Retracting justifications

To make a justification retractable, it is conjoined with an extra assumption which represents its defeasability. To retract the justification, the problem solver contradicts the conjoined assumption. This is the only reasonable way to retract a justification.

Although some TMSs permit direct retraction, few implement it correctly (correct means that all the labels of the datum will be as if the removed justification never existed). Retraction is inherently inefficient for all TMSs. (Direct justification retraction is isomorphic to LISP garbage collection.) The ATMS incorporates a correct but very inefficient direct retraction algorithm. Removing a justification from a node requires invalidating the labels of all the consequents (recursively) and recomputing them from correct labels. That part is easy and is akin to conventional garbage collection. The inefficiency arises if any of those recursive consequents is $\perp$. If so, an environment may have to be removed from the nogood database, and that can provoke an enormous amount of work. If a nogood has to be removed, the more specific nogoods which it subsumed have to be put back and every label of every node is potentially affected. For these reasons, directly removing a justifications is a bad idea.

4.10. Classes

It is often convenient to group nodes together into sets and treat each member identically. The architecture incorporates a notion of node set, or *class*. The node-class framework is not logically necessary, but it is a minor extension which greatly simplifies problem-solver-ATMS interactions. De Kleer, in [8], utilizes this framework to encode complex inference rules.

The node-class organization is intended to be very general. A node can be a member of any number of classes (possibly none), and a class can have any

number of members (possibly none). The membership of a node in a class has no relation to whether it holds in any environment. Unless a class has been specifically closed, nodes can be added to it at any time. It is not possible to remove a node from a class.

In many applications a node can be a member of at most one class. In such cases, the class is viewed as a variable and its nodes as values. The node $\gamma_{x=n}$ represents the fact that class x has value n, and is distinguished from the node $\gamma_{x=n}$ which represents the equation $x = n$. Such classes are used to represent the usual notion of variable in programming or constraint languages. In the usual constraint-language terminology [21], a cell is represented as a class and a value by a node. Unlike conventional constraint languages, a cell can simultaneously have many values. In most applications differing values for the same variable are inconsistent.

Suppose $X \in \{1, 2, 3\}$. A closed class is constructed for x, and is given three nodes: $\gamma_{x=1}$, $\gamma_{x=2}$ and $\gamma_{x=3}$. Each pair of these nodes are marked inconsistent, e.g., $\gamma_{x=1}, \gamma_{x=2} \Rightarrow \gamma_{\perp}$. $\Gamma_{x=n}$ refers to the assumption which justifies $\gamma_{x=n}$:

$$\Gamma_{x=n} \Rightarrow \gamma_{x=n} .$$

Classes are also used to represent patterns in assertional languages. A pattern represents the collection of its instances (i.e., nodes). Every assertion is a node, and every rule antecedent pattern is a class. Unlike constraint languages, a node can be a member of an arbitrary number of classes, as the same assertion match multiple-antecedent patterns. (See [8] for more details.)

5. Limitations Removed

The single-state problem. As the assumption-based approach does not require a notion of current global context, multiple, mutually contradictory, solutions may coexist. Thus, it is simple to compare two solutions.

Overzealous contradiction avoidance. The presence of two contradictory data does not terminate work on the overall knowledge state, rather only other data which depend on both contradicting data are removed. This is exactly the result desired from a contradiction—no more, no less. The nogood database guarantees that mutually contradictory data will never be combined. Said differently, the nogood database, in effect, provides the partitioning of the database into the consistent solutions.

Switching states is difficult. Changing state is now trivial or irrelevant. A state is completely specified by a set of assumptions. Problem solving can be restricted to a current context (i.e., a set of assumptions) or all states can be explored simultaneously. In either case, data obtained in one state are "automatically" transferred to another. For example, if $\langle x = 1, \{\{E, G\}\}, \{\ldots\}\rangle$ is deduced while exploring $\{B, C, E, G\}$, then $x = 1$ will still be present while exploring $\{B, D, E, G\}$.

The dominance of justifications. As assumptions, not justifications, are the dominant representational mode it is easy to compare sets of assumptions underlying data. For example, it is easy to find the datum with the most assumptions or the least; it is easy to determine whether the presence of one datum implies the presence of another (a implies b if every environment of a is a superset of one of the environments of b). Also, the justifications underlying a datum never change.

The machinery is cumbersome. The underlying mechanism is simple. There is no backtracking of any kind within the ATMS—let alone dependency-directed backtracking. The assumptions underlying a contradiction are directly identifiable. It is not necessary to explicitly mark data as believed or disbelieved, or signal the problem solver when the statuses of data change.

Checking for circular supporting justifications is unnecessary. Although it is simple to construct circular justifications, the basic ATMS mechanism will never mistakenly use it as a basis for support. Consider a database consisting of only two justifications: $a \Rightarrow b$ and $b \Rightarrow a$. This is represented by two nodes: $\langle a, \{\}, \{(b)\}\rangle$ and $\langle b, \{\}, \{(a)\}\rangle$. The circular justification has no effect as both nodes have empty labels. Suppose a receives an environment. This label propagates to b, and back to a, but the second update leaves the label unchanged terminating the propagation. Notice that even if the loop back to a were longer, any environment added by b to a would have to be a superset of one of a's. The point is that circularities cause no special difficulties for the ATMS label propagation mechanism.

Unouting. For those tasks requiring all solutions (or where there is only one solution), the ATMS completely finesses the earlier unouting problem. Unlike with a conventional TMS, the ATMS allows the problem solver to explore all solutions at once. There is no backtracking, no retraction, and no context switching. Thus, the previous unouting example is handled trivially:

(1) $x \in \{1, 2\}$

(2) $y \in \{1, 2\}$

(3) $x \neq y \oplus x = y$

(4) $z = (1000xy)!$

(5) $r = (100xy)!$

Consider the problematic ordering in which z is computed before the contradiction is detected. In this simple architecture, a contradiction within $\{\Gamma_{x=1}, \Gamma_{y=1}, \Gamma_{x\neq y}\}$ merely implies that any exploration of that state ceases, i.e., any inferences involving environments which contain $\{\Gamma_{x=1}, \Gamma_{y=1}, \Gamma_{x\neq y}\}$ are avoided. Work on environment $\{\Gamma_{x=1}, \Gamma_{y=1}, \Gamma_{x=y}\}$ continues as if the contradiction never occurred. Data derived from $\{\Gamma_{x=1}, \Gamma_{y=1}\}$ alone are automatically part of every superset, and hence are also part of $\{\Gamma_{x=1}, \Gamma_{y=1}, \Gamma_{x=y}\}$. There is never any question of rerunning rules.

Unouting problems also arise due to *implicit* context switches caused by alternative justifications for previously derived data. Consider a new example. The problem solver has proceeded until the following database state is reached:

$$\gamma_{x=1}: \langle x = 1, \{\{A\}\}, \{\ldots\}\rangle,$$
$$\gamma_{y=-x}: \langle y = -x, \{\{B\}\}, \{\ldots\}\rangle,$$
$$\gamma_{z=x}: \langle z = x, \{\{C\}\}, \{\ldots\}\rangle,$$
$$nogood\{A, B\}.$$

The problem solver immediately notices that it has a value for x which it could substitute into the equation $y = -x$, however the value would be computed under $\{A, B\}$ which is a nogood set so the substitution is not permitted. Now suppose the problem solver discovers the premise

$$\gamma_{z=1}: \langle z = 1, \{\{\ \}\}, \{\ldots\}\rangle.$$

The equation $z = x$ provides a different way of computing x:

$$\gamma_{x=1}: \langle x = 1, \{\{C\}, \{A\}\}, \{(\gamma_{z=x}, \gamma_{z=1})\ldots\}\rangle.$$

Now $\langle y = -1, \{\{B, C\}\}, \{(\gamma_{x=1}, \gamma_{y=-x})\}\rangle$ is derivable. Thus, $\gamma_{x=1}$ has no believed consequents under its original label $\{\{A\}\}$ and has consequent $y = -1$ under its new label $\{\{C\}, \{A\}\}$. The unouting problem is whether to work on $x = 1$ again after the second derivation is found for it.

De Kleer [5] and Williams [22] suggest an inelegant scheme for dealing with multiple justifications for the same datum. Suppose a second derivation for the same datum is discovered, the old environment being E_{old} and the new one E_{new}. There are four possible cases. If E_{old} is the same as E_{new}, then the new derivation holds in exactly those contexts where the old one was, so no new information can be gained by having the problem solver reexamine the datum. If E_{old} is a proper subset of E_{new}, then the new datum is a more specific instance which holds in a subset of the contexts where the old one did so the datum can also be ignored. On the other hand if E_{new} is a proper subset of E_{old}, the new instance is a more general and might hold in *more* contexts than the old one. If the two environments are incomparable the new instance might hold in other contexts than the old one. In both these latter cases the problem solver is reinvoked on the datum, deriving all of its consequents, rederiving the old consequents and in addition deriving the consequents which could not be derived in the old environment. This process recurs. Note that this recursion might descend many levels before a genuinely new piece of problem-solving work is done (if ever). ART handles this problem in an equivalent, and thus

problematic, manner by creating multiple copies of the datum if necessary. De Kleer presents a problem-solver-TMS protocol in which no piece of problem solving need ever be done twice in [8].

6. Implementation Issues

To avoid the ATMS being the dominant consumer of resources of the overall problem solver, great care must be taken in the choice of the representations and algorithms. The following is a description of the basic design choices and some of their alternatives.

Implemented straightforwardly, the ATMS would be too slow, spending all of its time performing set unions and subset tests. The implementation employs two representation ideas to minimize both the cost and the number of set operations. Sets are represented by bit-vectors: each assumption is assigned a unique position so that a one bit in a position indicates the presence of the corresponding assumption in the set. The union of two sets is computed by or'ing the bit-vectors. Set1 is a subset of set2 if the result of and'ing the bit-vector of set1 with the complement of set2 is zero.

As every environment has to be checked whether it contains a contradiction, the nogood database is kept as small as possible. Therefore, whenever a contradiction is discovered, its more specific instances (i.e., its supersets), if any, are removed from the nogood database.

Most subset tests occur when checking whether environments contain nogoods. Almost all such subset tests can be avoided by having a unique data structure for each environment. Whenever a union operation produces a new environment it must still be checked to see whether it contains a contradiction. However, this unique data structure ensures that this test be done once per environment, not once per union operation. This is implemented with a hash table using bit-vectors as keys (here we see another advantage of the bit-vector representation—it is very easy to compute the hash for a bit-vector). The only expensive operations thus are the creation of a new environment and the discovery of a new contradiction. Note that as supersets of nogoods have few interesting properties it is unnecessary to create a full environment structure for them.

Contradictions involving less than three assumptions are common and are best handled with special case mechanisms. A contradiction of the empty environment indicates an error which should be reported to the user without performing any ATMS operations. A contradiction of length one indicates an individual assumption has become inconsistent. Each environment containing this assumption becomes inconsistent. Furthermore, every ATMS operation on assumptions first checks to see whether any of the supplied assumptions are inconsistent. If so, the operation is ignored. The efficient handling of binary contradictions requires a more specialized representation. Each assumption is

associated with the set of assumptions it is inconsistent with. Like label environments, these sets are represented by bit-vectors. Every ATMS operation which attempts to create an environment first checks whether its results would violate a binary contradiction. If so, the operation fails, and the environment is not created. Although this mechanism slows down all environment creation operations, the reduction in the number of nogoods and environments greatly improves overall efficiency.

As a side-effect of the representation of binary nogoods, oneof disjunctions are represented at no cost. Without this efficient representation, n^2 nogoods would have been required to represent a oneof disjunction of size n.

In most ATMS applications most of the environment unions involve one environment of length one. Such union operations can be highly optimized. The optimization is based on viewing the union as adding a single assumption to the environment. This optimization uses three caches. The first cache, associated with each assumption, is the set of minimal nogoods of size three or larger in which the assumption appears. When an assumption is added to an environment, the consistency check of the resulting environment need only be checked against this set. The second cache, associated with each environment, is the set of assumptions it is inconsistent with, represented as a bit-vector. The third cache, also associated with each environment, associates each added assumption with the resulting consistent environment. These three caches, combined with the environment hash table and the binary contradiction optimization, make adding an assumption to an environment very fast.

The set of all environments and the set of all nogoods should be organized by length. A new environment needs only be tested against every nogood smaller than it. Likewise every new contradiction needs only be tested against every environment larger than it.

The label-update algorithm should not be applied to true (having an empty environment) or false (marked contradictory) nodes. Instead, label updates of true nodes should be ignored, and updates of false nodes should be computed incrementally and added directly to the nogood database.

Nodes can be inferred to be contradictory. If all but one of the antecedents of a contradictory node are true, then that one antecedent is necessarily false as well. This requires a second, but extremely simple propagator which follows antecedent links of justifications.

Assumptions consume two valuable resources: bit-vector positions and environment hash-table positions (that contain the assumption). Therefore it is worthwhile garbage collecting assumptions. An assumption can be garbage collected if either all its consequent nodes are true (i.e., hold universally) or false (i.e., is contradictory), or if the singleton environment containing the assumption is contradictory. In my experience, typically 30% of the assumptions can be garbage collected. When the assumption is garbage collected, all

environments which include it are removed from the environment hash table and nogood database, and the bit-vector position representing it is recycled. The garbage collector is run whenever the environment hash table is grown and rehashed. Some care must be taken that assumptions which have not yet been integrated into all their justifications are not accidentally garbage collected. The problem solver must indicate when an assumption becomes garbage collectible.

Bit-vectors are only worth using because operations on them are efficiently executed by computer hardware. The environments could also be represented as ordered lists. Union and intersection would then take linear time of order of the size of the environments. With bit-vectors the time is linear but each test is of order of all the assumptions ever considered and thus is potentially exponential.

Caching the results of all subset tests and union operations is not worth it. The bit-vector representation combined with the hash table is so efficient that the overhead of maintaining the cache is more expensive than the operation itself.

Representing all the subset relations in a dag (directed acyclic graph) is not worth it. Even with the dag it is not worth using the dag for a subset test because bit-vector operations are so fast. Sweeping out all superset good environments or sweeping out all subset nogood environments, however, is much faster. Unfortunately, the time and space resources of maintaining the dag overwhelms this advantage.

The following are untried possibilities.

A dag consisting only of unions. Instead of caching all subset tests, one can only cache the subset links implied by union operations actually required in the problem solving. To determine whether a new environment contains a contradiction one can look at the supersets of the antecedents which are contradictory. This type of scheme is used in ART [22, 23].

A discrimination net for nogoods. In my experience, relatively few of the environments actually encountered are nogood, thus it makes sense to optimize the subset test at the expense of a complex representation.

All the set operations are a result of updating the labels of nodes. However, unless the node is examined for some reason, there is no reason to update its label. Thus node labels could be updated only if needed. This requires a second propagator which propagates 'update' messages backwards along justification links. However, the labels for contradictory nodes must always be updated otherwise contradictions will be missed.

Introducing these efficiency techniques extends the data structures required for assumptions and environments (no changes are necessary to nonassumption nodes): In addition to the slots for a basic node, an assumption has three additional slots.

- Position: Bit-vector position corresponding to this assumption.
- Contras: Bit-vector of other assumptions it is inconsistent with.
- Nogoods: Minimal nogoods of length three or greater in which it appears.

An environment has three additional slots:

- Contras: Bit-vector of other assumptions it is inconsistent with.
- Cache: A cache of the results of adding assumptions to this environment.
- Count: Number of assumptions.

7. Summary

The ATMS is given a set of assumptions A, a set of nodes N ($N \supset A$), and a set of justifications J in terms of N (which includes A). The triple $\langle A, N, J \rangle$ implicitly defines a set of contexts where a context is defined as a consistent (i.e., not admitting $\gamma_\perp$) set of assumptions ($A' \subset A$) and nodes ($N' \subset N$) derivable from A' using J. Given k assumptions there are conceivably 2^k contexts. The ATMS never explicitly computes these contexts. The contexts provide a convenient concept with which to define what the ATMS actually does. A node's label is a minimal description of the characterizing environments of the contexts within which it appears. A node is a member of every context having a characterizing environment which is a superset of one of the label's environments.

As a consequence the ATMS has four important properties. These properties are worth highlighting because they both illustrate the utility of the ATMS to problem solving and because care must be taken to preserve them when more complex axioms are permitted as in J (in [7]).

Most fundamentally, *label consistency* ensures all inconsistent environments are identified and defined not to have contexts and do not appear in node labels.

Label soundness ensures that no context will contain a datum it should not. Thus, nodes which hold in no context will have empty labels. This property is important to problem solving because it ensures that the problem solver will not mistakenly work on irrelevant nodes which do not hold in any context.

Label completeness guarantees that every context will contain every datum it should. Thus, all nodes which hold in any context have nonempty labels. This property is important to problem solving because it ensures that the problem solver can easily identify all relevant nodes (i.e., those which hold in at least one context).

Label minimality states that each node label has the fewest disjuncts and that each disjunct has the fewest assumptions. This property is important to the problem solver because it ensures that changing any label assumption affects the node's status. This property is true by definition for the basic ATMS, but becomes an important consideration for dealing with disjunction axioms.

Label consistency and minimality are also important for ATMS efficiency.

Although label soundness allows the problem solver to ignore irrelevant

nodes, this does not guarantee that these nodes will ultimately be irrelevant. Conversely, although label completeness enables the problem solver to identify all relevant nodes, this does not guarantee that these nodes will ultimately be relevant. If the problem solver could be guaranteed to work only on ultimately relevant nodes, the entire problem-solving process should be written as an efficient straightforward algorithm. All the ATMS can ensure is, given the justifications and assumptions *so far*, that these properties hold. With the addition of every justification, the set of contexts may change and nodes that were irrelevant become relevant and vice versa.

ACKNOWLEDGMENT

I have received tremendous technical help and encouragement from Daniel Bobrow, Ken Forbus, Brian Williams, and Benjamin Grosof. David McAllester helped clarify the differences between the ATMS and conventional TMSs. Greg Clemenson, Randy Davis, David Etherington, Richard Fikes, Jeff Finger, Matthew Ginsberg, Kris Halvorsen, Ken Kahn, Paul Morris, Ray Reiter, Peter Struss, Chuck Williams, provided valuable discussions. Many of their ideas appear throughout this paper. Lenore Johnson drew the figures. I cannot thank them all enough.

REFERENCES

1. Brown, J.S., Burton R.R. and de Kleer, J., Pedagogical, natural language and knowledge engineering techniques in SOPHIE I, II and III, in: D. Sleeman and J.S. Brown (Eds.), *Intelligent Tutoring Systems* (Academic Press, New York, 1983) 227–282.
2. Clocksin, W.F. and Mellish, C.S., *Programming in Prolog* (Springer, Berlin, 1981).
3. de Kleer, J., Local methods of localizing faults in electronic circuits. Artificial Intelligence Laboratory, AIM-394, MIT, Cambridge, MA, 1976.
4. de Kleer, J., Causal and teleological reasoning in circuit recognition, Artificial Intelligence Laboratory, TR-529, MIT, Cambridge, MA, 1979.
5. de Kleer, J., Choices without backtracking, in: *Proceedings Fourth National Conference on Artificial Intelligence*, Austin, TX (1984) 79–85.
6. de Kleer, J., How circuits work, *Artificial Intelligence* **24** (1984) 205–280.
7. de Kleer, J., Extending the ATMS, *Artificial Intelligence* **28** (1986) 163–196 (this issue).
8. de Kleer, J., Problem solving with the ATMS, *Artificial Intelligence* **28** (1986) 197–224 (this issue).
9. de Kleer, J. and Brown, J.S., A qualitative physics based on confluences, *Artificial Intelligence* **24** (1984) 7–83.
10. de Kleer, J. and Sussman, G.J., Propagation of constraints applied to circuit synthesis, *Circuit Theory and Applications* **8** (1980).
11. de Kleer, J. and Williams, B.C., Diagnosing multiple faults, *Artificial Intelligence*, submitted.
12. Doyle, J., A truth maintenance system, *Artificial Intelligence* **12** (1979), 231–272.
13. Martins, J.P., Reasoning in multiple belief spaces, Department of Computer Science, Tech. Rept. No. 203, State University of New York, Buffalo, NY, 1983.
14. McAllester, D., A three-valued truth maintenance system, S.B. Thesis, Department of Electrical Engineering, MIT, Cambridge, MA, 1978.
15. McAllester, D., An outlook on truth maintenance, Artificial Intelligence Laboratory, AIM-551, MIT, Cambridge, MA, 1980.
16. McAllester, D., Reasoning utility package user's manual, Artificial Intelligence Laboratory, AIM-667, MIT, Cambridge, MA 1982.
17. McDermott D., Contexts and data dependencies: a synthesis, *IEEE Trans. Pattern Anal. Machine Intelligence* **5** (3) (1983).

18. Reiter, R., A logic for default reasoning, *Artificial Intelligence* **13** (1980) 81–132.
19. Rulifson, J.F., Derkson, J.A. and Waldinger, R.J., QA4: a procedural calculus for intuitive reasoning, Artificial Intelligence Center, Tech. Note 73, SRI, Menlo Park, CA, 1972.
20. Stallman, R.M. and Sussman, G.J., Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis, *Artificial Intelligence* **9** (1977) 135–196.
21. Steele, G.L., The definition and implementation of a computer programming language based on constraints, Artificial Intelligence Laboratory, TR-595, MIT, Cambridge, MA 1979.
22. Williams, C., ART the advanced reasoning tool—conceptual overview, Inference Corporation, 1984.
23. Williams, C., Managing search in a knowledge-based system, unpublished, 1985.

Received April 1985; revised version September 1985