Focusing the ATMS

Kenneth D. Forbus
Qualitative Reasoning Group, University of Illinois
1304 W. Springfield Avenue, Urbana, Illinois, 61801
Johan de Kleer
Xerox Palo Alto Research Center
3333 Coyote Hill Road, Palo Alto CA 94304

Abstract

Many problems having enormous search spaces can nevertheless be solved because only a very small fraction of that space need be traversed to find the needed solution(s). The ability of assumption-based truth maintenance systems to rapidly switch and compare contexts is an advantage for such problems, but existing ATMS techniques generally perform badly on them. This paper describes a new strategy, the *implied-by* strategy, for using the ATMS that efficiently supports problem solving in domains which are infinite and where the inference engine must retain tight control in the course of problem solving. We describe the three mechanisms required to implement this strategy: *focus environments, implied-by consumers*, and *contradiction consumers*. We compare the implied-by strategy to previous ATMS strategies, and demonstrate its effectiveness by performance comparisons in two simple domains. Finally, we discuss the implications for parallel processing and future ATMS design.

1 Introduction

For many problems, the search space is vastly larger than the subset which needs to be explored to find an acceptable solution. Such problems include the design of engineered systems, interpreting data from multiple sources, solving textbook physics problems, and scientific discovery. Problem solvers for such domains must carefully shepherd their resources, narrowly focusing attention on only a small number of alternatives, while keeping track of the possibility that their current focus could be wrong.

Assumption-based truth maintenance provides significant advantages for such problems, due to compact representation of contexts which allows rapid context-switching and comparison between alternatives. However, current ATMS/inference-engine interfaces are oriented towards finding all (or many) solutions, making them unsuitable for such problems. Two central issues in such interfaces are (1) where control resides and (2) how rules are scheduled for execution. The simplest ATMS-based search technique for generating global solutions is *interpretation construction*, a form of backtrack search. It assumes that (a) all sets of alternatives are known in advance, and (b) no new information is added during the course of problem solving. These assumptions are false for the kinds of domains described above. The sets of choices that are relevant in a design, for instance, depend in part on earlier choices in the design, and the set of potential choices is far too large to elaborate explicitly before beginning a search. The existing consumer architecture [3] schedules a rule for execution whenever all its antecedents simultaneously hold in some consistent context. For the domains of interest here, this strategy leads to much wasted effort. Consider an intelligent design aid for VLSI. If the initial design called for CMOS, the program might begin elaborating and exploring the various alternatives under this assumption. If external factors force the design to use gallium arsenide instead, it should stop working on the CMOS version of the design. The CMOS design has not become inconsistent, it has simply become irrelevant (at least for the time being). Yet a program based on the simple consumer architecture will continue to pursue both.

The idea of *assumption-based dependency-directed backtracking* (ADDB) outlined in [4] provides some help, but not enough. Representing alternatives via explicit, conditional control disjunctions allows new alternatives to be added during a search, unlike interpretation construction. But control still resides in the ATMS, with no provision for exploiting domain-specific information availible to the inference engine. Control disjunctions allow rules to be scheduled more efficiently, since only rules whose antecedents are consistent with the currently believed set of control assumptions are executed. However, this strategy can still lead to inefficiencies, since rules which are irrelevant but consistent can still be executed. While these restrictions are tolerable for many problems, the techniques we describe here transcend them. We provide a more flexible and general ATMS/inference-engine interface which minimizes total problem-solving work in cases where only a single (or small number) of solutions are sought.

In this paper we describe a strategy, called the *implied-by* strategy, for building ATMS-based problem solvers which can efficiently explore large search spaces. Three mechanisms are needed to support this strategy. *Focus environments* provide the interface between the inference engine's center of attention and the ATMS. *Implied-by consumers* provide an antecedent rule mechanism that respects the inference engine's focus. *Contradiction consumers* provide a signaling mechanism for the ATMS to inform the inference engine about inconsistencies. Each mechanism is a straightforward extension of existing ATMS technology, but together they combine to provide the power we seek.

2 ATMS Background

We assume that the problem solver consists of the ATMS and an *inference engine*, whose job is to control the problem-solving process, in part by deciding what assumptions and justifications are to be fed to the ATMS. We also assume that much of problem solver's knowledge is encoded in the form of rules. The triggering of rules on data is handled in the inference engine. When rules are triggered, *consumers* are created and passed to the ATMS

to allow it to execute the rules under appropriate circumstances. Such rules might implement modus ponens, or apply an axiom-schema to assert the consequences of a definition.

Briefly, the basic ATMS is characterized as follows. Every datum the problem solver reasons about is assigned an ATMS *node*. The problem solver designates a subset of the nodes to be *assumptions* — nodes which are presumed to be true unless there is evidence to the contrary. Every important derivation made by the inference engine is recorded as a *justification*:

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

Here $x_1, x_2, \ldots$ are the antecedent nodes and n is the consequent node. An ATMS *environment* is a set of assumptions. A node n is said to hold in environment E if n can be propositionally derived from the union of E with the current set of justifications (viewed as propositional Horn clauses). An environment is inconsistent (called *nogood*) if the distinguished node $\perp$ (i.e., false) holds in it.

The ATMS is incremental, receiving a constant stream of additional nodes, additional assumptions, additional justifications and various queries concerning the environments in which nodes hold. To facilitate answering these queries the ATMS maintains with each node n a set of environments $\{E_1, E_2, \ldots\}$ called its *label*. Each node's label has four properties:

1. n holds in each E_i.
2. E_i is not nogood.
3. Every environment E in which n holds is a superset of some E_i.
4. No E_i is a proper subset of any other.

Given the label data structure the ATMS can efficiently answer the query whether n holds in environment E by checking whether E is a superset of some E_i.

Several ways have been developed to organize problem-solvers around an ATMS. These techniques differ primarily in where control resides and what discipline is used to control rule executions. The simplest strategy is the *INTERN* strategy, where a rule is executed as soon as its antecedents are bound, whether or not they can be consistently believed together at that time. This strategy makes sense when rules are cheap, much of the search space is going to be explored, and all consistent solutions are sought. In carefully crafted programs, this strategy can be very efficient [7; 8]. However, it is unthinkable for infinite search spaces, and for general problem solving is extremely inefficient.

The next strategy is the *IN* strategy, where execution of a rule is postponed until the union of its antecedents can be consistently believed. This is the strategy of the consumer architecture described in [3]. While more controlled than the INTERN strategy, it can still be arbitrarily inefficient when only a handful of solutions is sought in a space which is largely consistent.

An important variation of the IN strategy is to use dependency-directed backtracking (the ADDB strategy, mentioned above). The inference engine informs the ATMS of its interests by asserting explicit *control disjunctions*. The ATMS maintains a focus environment consisting of one assumption from each control disjunction. Rules are executed only when the union of their antecedents is consistent with this focus. Should the focus become contradictory, the ATMS does not inform the inference engine but performs dependency-directed backtracking until a consistent new focus is found. Although effective for some tasks, this mechanism has two limitations. First, it can be "distracted" by extraneous information, and thus is ill-suited for infinite domains (See Section 4.2). Second, it is too inflexible for general problem solving. The ATMS is a domain-independent module, hence it cannot have as much information about the particular task demands as the inference engine does. The choice of focus, what to do when it becomes inconsistent, and what to try next, are more properly the concerns of the inference engine than an automatic backtracking scheme.

3 The Implied-by Strategy

We assume the inference engine has some notion of task to organize its activities. This can be a node in an AND/OR tree, agenda item, etc. For simplicity, this discussion assumes that the inference engine works on only one task at a time (we mention more general strategies in Section 5).

Our strategy associates with each task a *focus environment*, the set of assumptions underlying it. These assumptions include both data of the problem and control assumptions which indicate why the task is reasonable. In a natural deduction system, for instance, the focus environment can include statements about what is being sought, propositions assumed by the user, and propositions assumed in the course of a proof.

When the problem solver begins to work on a task, it signals the ATMS that the environment associated with that task is now the focus. We define a new class of consumers, *implied-by consumers*, which are to be executed only when the union of their antecedents is implied by the current focus. By creating implied-by consumers, the inference engine constrains the ATMS to respect its choice of focus. All new information generated by the rules is a consequence of the focus, and hence likely to be relevant. Running the rules may provide the problem solver all or some of the information it needs to carry out that task. Every time the problem solver switches to a new task, the focus environment is changed accordingly.

Notice that including data assumptions in the focus is essential for the implied-by strategy to work. This is different from ADDB, which only has elements from control disjunctions in its focus. By only executing consumers which are implied by the focus, rather than consistent with it, our strategy provides finer control over problem-solving. Furthermore, ADDB requires that all assumptions appearing in a control disjunction must be defined before that disjunction is asserted. in a particular control disjunction have to be defined when the control disjunction is asserted. In the implied-by strategy, assumptions are only created when needed, thereby creating fewer assumptions and thus fewer chances for distraction.

Sometimes, rules are used to check consistency of a proposed solution or set of data. A design system, for instance, must ascertain whether a proposed device will satisfy its cost/performance constraints. Such rules may install justifications that result in the focus environment becoming

inconsistent. What happens at this point must depend on the task. In some cases, the inconsistency could indicate that user-supplied data is faulty ("the patient's temperature is 986 degrees F"). In other cases, the inference engine could be deliberately attempting to generate a contradiction as part of an indirect proof. The inference engine indicates what should be done by installing *contradiction consumers* on focus environments. A contradiction consumer is associated with a particular environment. If an environment becomes contradictory, all contradiction consumers associated with it are executed.

Contradiction consumers extend the problem-solver's power by providing an "interrupt" mechanism, the logical equivalent of a divide-by-zero interrupt in a numerical program. If the purpose of the task was to find a contradiction, that contradiction can then be analyzed and appropriate steps taken (such as installing a justification with the conflicting assumption discharged). If a contradiction was unexpected, then that task might be deactivated, and another task selected.

Contradiction consumers are executed whenever a focus environment becomes contradictory, whether or not it is the problem solver's current focus. This provides the inference engine with maximum information as quickly as possible. Consequently, these consumers should only perform changes on the inference engine's internal representation of the particular task which installed them. For instance, if a contradiction is unexpected a useful default action is to mark the task as unachievable. Other mechanisms in the problem solver must decide what other effects this change in status should have. The reason is that focus environments for different tasks often overlap – if task T_2 is spawned by task T_1, for instance, it typically is the case that $focus(T_1) \subseteq focus(T_2)$. Thus the control component of the inference engine must be able to refocus on a new task when a number of existing tasks become inconsistent simultaneously. Some control mechanisms provide this ability more easily than others – it is easy using priority queues or agendas, somewhat harder using a stack-model, and can be very complicated with arbitrary lisp code.

A focus environment might be discovered to be inconsistent at any time. We suspend execution of implied-by consumers when the focus is contradictory, deferring further action until the inference engine makes a wiser choice of focus.

These mechanisms are implemented in a problem-solving language we have developed called **ATMoSphere**. **ATMoSphere** is a descendent of **DEBACLE** [6], which is a descendent of **RUP** [10], which is a descendent of **AMORD** [1]. **ATMoSphere** contains pattern-directed rules which are matched antecedently. Three conditions for rule execution are provided, corresponding to the three strategies described above: `:intern`, where rules are executed on matching; `:in`, where rules are executed when their antecedents can be believed together consistently, and `:implied-by` rules.

4 Examples

We illustrate the implied-by strategy by outlining how to use it to build problem solvers for two kinds of problems. The first, natural deduction, shows that by using the implied-by strategy ATMS-based problem solvers can indeed work efficiently in infinite domains. The second, cryptarithmetic, shows that the implied-by strategy can be more efficient even in domains where the other ATMS strategies are applicable. These systems have been fully implemented, and statistics from our experiments with them are presented. Both problem solvers are implemented on top of a common control component, which we describe first.

4.1 A Control Component

AND/OR trees are a classical way to organize problem-solving activity. We built a simple inference engine using AND/OR trees which interacts with an ATMS using the implied-by strategy. A problem is specified by a collection of initial assumptions and a goal, which forms the root of the AND/OR tree. We refer to elements of these trees as *ao-nodes*. Each ao-node represents an inference engine task. A problem is solved by expanding from the root ao-node until either (1) resource bounds are exceeded, (2) no further expansion is possible, or (3) the root goal is satisfied. The expansion is carried out by a monitor program, which uses a scoring mechanism (programmable) to determine which ao-node to attempt expanding next.

So far, we have described a traditional AND/OR scheme. This scheme is integrated with the ATMS as follows. Each ao-node is created with a focus environment. Its focus is that of its parent, typically extended by an additional assumption. A contradiction consumer is created for the ao-node's focus whose default behavior is to deactivate the task associated with that ao-node.

When an ao-node is expanded, all implied-by consumers relevant to its focus are executed. These consumers have three functions. First, they detect inconsistencies, and thus may rule out the current focus. Second, they make "obvious" inferences (e.g., modus ponens), which produce the desired answer. Third, the rules can make *suggestions* about what to try in order to achieve the current ao-node's goal, if necessary. Thus, like the other ATMS-based problem-solvers, much of the knowledge is encoded in terms of pattern-directed rules. Like other strategies, we assume that executing individual rules takes finite time, and that executing combinations of rules will always take finite time. Each activity suggested by a rule must by itself take finite effort. The difference is that we allow for the possibility that following up on *all* suggestions made by the rules could require infinite effort. Any activity which could lead to infinite behavior (such as suggesting that, given an integer, we consider its successor since it, too, is an integer) must be proposed as a suggestion. The ao-node being expanded then becomes an OR ao-node, with each suggestion comprising a subgoal. Thus the choice of what to do next (and any responsibility for infinite loops) resides in the inference engine, not the ATMS.

What happens when an ao-node is expanded depends on the type of goal. The goal type **SHOW-ANY** causes the ao-node to be marked as an OR node, and creates ao-nodes for each of the arguments. **SHOW-ALL** works similarly, except the expanded node is marked as an AND node. Expanding **SHOW-ANY** and **SHOW-ALL** ao-nodes does not execute rules, it just makes explicit the logic which the monitor must follow.

Figure 1: Rules for introducing and eliminating implications

The syntax of ADB rules is similar to that of [1]: `(rule` *condition triggers . body* `)`. A *trigger* is either an *expression* or *expression* `:var` *variable* where *variable* is bound to the the entire expression. These rules implement conditional elimination (modus ponens) and introduction. The first rule simply carries out modus ponens. The second rule suggests that showing `?p` is a useful thing to do if `?q` is sought and you know `(implies ?p ?q)`. The third rule provides a way of introducing conditionals; it suggests that a new context be sprouted (`try-box`), in which `?p` is assumed, `?q` (or a contradiction) is sought, and if found, then `(implies ?p ?q)` is justified.

```
(rule :implied-by ((implies ?p ?q) :var ?f1
                   ?p)
      (rjustify ?q (?f1 ?p) :Modus-Ponens))
 (rule :implied-by ((show ?q) :var ?f1
                    (implies ?p ?q) :var ?f2)
       (rjustify (suggest-for ?q (show ?p))
                 (?f1 ?f2) :BC-Modus-Ponens))

(rule :implied-by ((show (implies ?p ?q)) :var ?f1)
      (rjustify (suggest-for (implies ?p ?q)
      (try-box ?p ?q (implies ?p ?q) CI))
               (?f1) :BC-CI)))
```

The real work is done by the goal type **SHOW**, which asks if a fact holds. The monitor expands an ao-node whose goal type is **SHOW** with the following procedure:

1. Check if the goal is already true. If so, mark ao-node as succeeding and close it.
2. Execute rules implied by the ao-node's focus.
3. Check if goal is now true. If so, mark ao-node as succeeding and close it.
4. Fetch suggestions for ao-node's goal.
   (a) If none, mark ao-node as failing and close it.
   (b) Spawn a child ao-node for each suggestion, marking the current ao-node as an OR node.

Application programs can add goal types by providing functions to carry out the expansion.

4.2 Natural Deduction

The rules for natural deduction, even for propositional problems, permit both the number and size of formulas to grow without bound. In this section, we show how the implied-by strategy is used to control the potentially infinite search for a proof. The system we describe here has been verified with numerous examples from introductory logic textbooks.

The particular natural deduction system we implement is based on [9]. The metalinguistic predicate **show** indicates interest in the proposition which is its argument. The rules are organized around the introduction and elimination of each kind of connective. The rules for implication are shown in Figure 1 as an illustration.

Table 1: Performance versus scheduling technique
This table shows, for a sample of six natural deduction problems, the number of rules executed under different scheduling techniques. The small differences indicate that for this class of problem, focus environments and contradiction consumers are the principal source of constraint in the implied-by strategy.

	Ex 1	Ex 2	Ex 3	Ex 4	Ex 5	Ex 6
Implied-By	7	54	17	17	47	60
In	11	54	17	18	41	60
Intern	11	64	17	18	41	76

When expanding an ao-node, the AND/OR monitor fetches all **suggest-for** assertions which match the current goal and are implied by the current focus. The focus for each new ao-node includes the assumptions of the focus of its parent, plus the control assumption corresponding to its goal. New data assumptions are introduced by the special goal type `try-box`. In the Kalish & Montague natural deduction system, a graphical notation involving boxes is used to depict a tree of assumptions on paper. Boxes can only be introduced by particular rules, and upon being completed, conclusions resting on that assumption cannot be accessed. We obtain the same effect by producing a daughter ao-node whose focus includes the data assumption in question, and installing two procedural hooks. The first is a procedure which is run whenever the ao-node is closed successfully, and installs the discharged justification. The second is a contradiction consumer which installs the appropriate discharged justification if the focus becomes contradictory. This consumer implements the notion that anything follows from a contradiction – one could always get the desired conclusion in one step by using an indirect proof. (Indirect proof is also implemented as an explicit proof rule, using this same contradiction consumer.)[1]

We used this system to address the following question: Which of the three mechanisms (contradiction consumers, implied-by rules, or focus environments) used to implement the implied-by strategy, provides the most leverage? It is easy to change all the rules to use either the IN or INTERN conditions, and the results of doing so are summarized in Table 1. In this class of problem, the IN and implied-by conditions are more or less equivalent, and both are better than INTERN (as would be expected). Two questions arise: (1) Why does the IN condition perform slightly better in one case? and (2) Why does the choice of rule condition make so little difference here?

The answer to the first question relies on a feature our problem solver shares with most simple systems – rules are executed until the queue is exhausted. Since simpler

[1] The search strategy followed by the inference engine is determined by the ao-node scoring function. Here, we use a simple measure which multiplies the depth of the ao-node times the depth of the expression, times a factor indicating relative difficulty of the type of goal. The first term biases the search toward shorter proofs, the second biases it towards simpler subgoals, and the third biases it against introducing assumptions. Two resource limitations were imposed, a bound on total number of ao-nodes and a bound on the maximum size of focus environments. The former prevents infinite searches, the latter restricts the size and complexity of proofs which are acceptable.

Table 2: Immunity to extraneous data
This table shows the number of rules executed as a function of triggering condition, like before, but with irrelevant assumptions added to each problem. The IN and INTERN conditions are sensitive to extra data, whereas the number of rules executed for the implied-by condition is the same. This illustrates that the implied-by strategy is relatively immune to distraction from extraneous information.

	Ex 1	Ex 2	Ex 3	Ex 4	Ex 5	Ex 6
Implied-By	7	54	17	17	47	60
In	27	71	33	34	58	78
Intern	27	81	33	34	58	94

foci are examined before more complicated ones (the focus of a child ao-node is always larger than the parent), this means contradictions will be discovered as early as possible – in some cases earlier than they would in the implied-by condition. For more realistic problems this is clearly not a reasonable technique [5]. If there are resource bounds within the execution of a task (as opposed to just bounds on the number or size of tasks), the IN condition simply does not provide enough fine-grained control. The ability of the implied-by strategy to guarentee that every rule execution is relevant to the chosen task would be essential in such cases.

The answer to the second question is slightly more subtle. The first reason is that, in all three cases, the introduction of new assumptions remains tightly controlled. Without contradiction consumers to close off lines of attack, and focus environments to tell the inference engine what is still feasible, arbitrary amounts of work could be wasted. Suppose, for instance, that two implications were being sought in order to use disjunction elimination[2]. If the search for one implication fails, the search for the other is fruitless. But a purely ATMS-based mechanism would probably continue to attempt proving it, since it would be consistent to do so.

The second reason is that these textbook problems are very simple – exactly the right amount of data required to solve the problem is provided, and no more. To show that this is the case, we added extraneous information, in the form of extra assumed propositions[3]. As can be seen in Table 2, the number of rules executed in the implied-by condition remain unchanged, while both IN and INTERN conditions waste effort on executing irrelevant rules. As the amount of irrelevant material grows larger, so does the degree of constraint contributed by the implied-by rules.

[2]The proof rule of disjunction elmination is

$$p \vee q, p \Rightarrow r, q \Rightarrow r \vdash r$$

The associated suggestion rule says that if we wanted r and had $p \vee q$, we should look for $p \Rightarrow r$ and $q \Rightarrow r$.

[3]We took the union of the premises of the original six examples, re-named the ground terms so as not to bias the solution, and assumed them before starting each problem in all conditions.

Table 3: Relative performance: Cryptarithmetic
This table shows the ATMS statistics for three simple problem solvers, one using the standard consumer architecture, one using ADDB, and one using the implied-by strategy, in solving the cryptarithmetic puzzle SEND + MORE = MONEY.

	Standard Consumers	ADDB Strategy	Implied-by Strategy
Assumptions	67	67	48
Rules	4163	1923	1454

4.3 Cryptarithmetic

The natural deduction system illustrates that the implied-by strategy allows us to use the ATMS effectively in infinite domains, something which no other strategy allows. Here, we show that even for finite domains, the implied-by strategy can be more efficient.

Cryptarithmetic is a standard example of a combinatorial problem [11]. An encoded arithmetic problem, such as

SEND + MORE = MONEY

is provided, and the problem is to find an assignment of digits to letters so that the sum comes out correctly. We implemented three problem solvers for these puzzles. The first uses standard ATMS technology, i.e., IN rules to install constraints between assumed letter/digit pairings, and interpretation construction to find all consistent combinations of pairings. The second uses ADDB. The third uses the implied-by strategy with the AND/OR tree monitor described earlier. In particular, the goal of each ao-node is to assign a digit to each of a list of letters. The constraints imposed by the column of the sum are enforced by implied-by rules. The focus environments consist of assumed letter/digit bindings, and contradiction consumers are installed to close the ao-node if the bindings are inconsistent. If an ao-node is not inconsistent, then it is expanded by suggesting possible bindings for the next letter in the list with all digits remaining. These suggestions respect the implications of the choices made so far, in that if the constraints imply a unique binding for a letter, that digit will be the only suggestion.

To provide a fair test we attempted to make these programs as alike as possible, given the radical differences in strategy. The relative performance of these two systems is shown in Table 3. Clearly, the ADDB and implied-by strategies are preferable, even though this kind of problem is just what the standard consumer architecture was designed for. The implied-by strategy comes out ahead of the ADDB strategy in total number of assumptions because it creates them only when needed, instead of building all of them in advance. Fewer assumptions means fewer consistent environments to trigger consumers, and hence fewer rules are executed under the implied-by strategy.

5 Discussion

This paper described the *implied-by* strategy, a new way to organize ATMS-based problem solvers. This strategy provides a way to exploit the advantages of an ATMS in infinite search spaces with extraneous data, a class of problem-solving situations which previously was viewed as unsuitable for ATMS usage. By providing implied-by rules, we allow much of the problem-solver's knowledge to be encoded in rules which will only be executed when relevant. By allowing the inference engine to select the focus of attention, we prevent the overall problem-solver from wandering aimlessly, executing rules which are consistent but not interesting. By providing contradiction consumers, we endow the inference engine with the ability to handle inconsistencies gracefully. Carried out correctly, this strategy prevents combinatorial explosions in the ATMS, putting responsibility back in the inference engine where it belongs.

There are several potentially useful extensions to this work. Although we have assumed only a single focus here, there is no difficulty in allowing multiple foci to facilitate parallelism. (Such an implementation might be an excellent candidate for coarse-grained parallel architectures.) Including intra-task resource bounds is slightly more difficult, since it requires tighter coupling between the ATMS consumer scheduler and the inference engine. However, this is clearly an important avenue to explore in scaling up the technology.

Originally, the ATMS was viewed as suitable mainly for finding all possible solutions. By introducing simple backtracking, some control over reasoning could be introduced, but the interface between ATMS and inference engine was restrictive and inflexible. In both cases, the overhead for many problems was still high compared to other TMS'. With the implied-by strategy, we have eliminated every extra overhead of the ATMS relative to other TMS' save one: Label updating. Relevant or not, label updating still occurs globally throughout the ATMS database. It is possible that this overhead, too, could be removed. Typically, any label whose environments contain no assumptions in common with the current focus is irrelevant to the current problem-solving activity. Adding such a test, and defering those updates until some overlap is discovered, might remove the last efficiency disadvantage of the ATMS in infinite domains. However, the bookkeeping required to avoid the label updates may outweigh the advantage of avoiding the updates.

6 Acknowledgments

Marianne Winslett provided useful comments. Pat Hayes suggested the name `ATMoSphere`. This research was supported in part by the Office of Naval Research, Contract No. N00014-85-K-0225, and by an NSF Presidential Young Investigator Award.

References

[1] de Kleer, J., Doyle, J., Steele, G.L. and Sussman, G.J., Explicit control of reasoning, in *Artificial Intelligence: An MIT Perspective,* edited by P.H. Winston and R.H. Brown, 93-118, (MIT Press, 1979). Also in: *Proceedings of the Symposium on Artificial Intelligence and Programming Languages,* 1977, Also in: *Readings in Knowledge Representation,* edited by R.J. Brachman and H.J. Levesque, 345–355 (Morgan Kaufman, 1985).

[2] de Kleer, J. "An assumption-based truth maintenance system", *Artificial Intelligence,* **28**, 1986.

[3] de Kleer, J., Problem solving with the ATMS, *Artificial Intelligence* **28** (1986), 197–224.

[4] de Kleer, J. and Williams, B.C., Back to backtracking: Controlling the ATMS, *Proceedings of the National Conference on Artificial Intelligence,* Philadelpha, PA (August 1986), 910-917

[5] Erman, L.D., Hayes-Roth, F., Lesser, V. R, and Reddy, D.R. "The Hearsay-II speech-understanding system: Integrating knowledge to resolve uncertainty", *Computing Surveys* **12**, 213–253, 1980.

[6] Forbus, K. "Qualitative Process theory", MIT AI Lab Technical report No. 789, July, 1984.

[7] Forbus, K. "The Qualitative Process Engine", Technical Report No. UIUCDCS-R-86-1288, December, 1986.

[8] Forbus, K. "`QPE`: A study in assumption-based truth maintenance" to appear, *International Journal of AI in Engineering,* 1988.

[9] Kalish, A., and Montague, R. *Logic: Techniques of formal reasoning* 2nd Edition. Harcourt Brace. 1980.

[10] McAllester, D. "Reasoning Utilities Package, Version One", MIT AI Laboratory Memo No. 667, April, 1982.

[11] Newell, A. and Simon, H. A. *Human problem solving.* Englewood Cliffs, N.J.: Prentice-Hall. 1972