# A Hybrid Truth Maintenance System

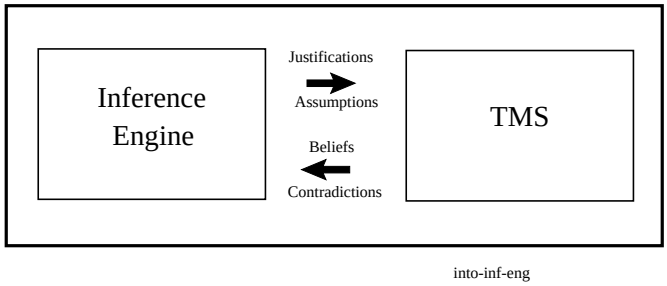
**Johan de Kleer**
Xerox Palo Alto Research Center
3333 Coyote Hill Road, Palo Alto CA 94304 USA
email: dekleer@xerox.com

December 22, 1994

**Abstract**

A variety of Truth Maintenance Systems (TMSs) have become widely used in AI. Most TMSs can answer the same queries, but different TMS algorithms require dramatically varying computational resources to do so. One dimension along which TMS performances differ is whether they are optimized to work best with a single context (e.g., a JTMS or LTMS) or whether they function best for multiple contexts (e.g., an ATMS). In this paper we present a new TMS algorithm, called a hybrid TMS (HTMS), which combines aspects of both approaches. The HTMS algorithm maintains partial ATMS labels; however, unlike an ATMS these labels are only sufficient to answer queries with respect to a restricted set of focus environments the inference engine has supplied it. We have used the new HTMS on a wide variety of problem-solving tasks and it shows an exponential performance improvement over the ATMS in most cases.

Figure 1: Problem solver = Inference Engine + TMS




# 1 Introduction

A variety of truth maintenance systems (TMSs) have become widely used in AI as components of overall problem-solving systems (see Figure 1). One of the objectives of this framework is to minimize the total computational cost (of TMS plus Inference Engine) yet provide the functionality that the problem-solving task demands. The partitioning of concerns between the TMS and the inference engine is quite delicate. If the TMS is given too large a share of the task, (e.g., the entire task is encoded in terms of justifications and assumptions), it performs poorly resulting in unnecessary combinatorial explosion. If the TMS is given too little of the task, then the benefits of the using a TMS are lost. This paper explores three dimensions of the partitioning of concerns: the type of TMS interface, the algorithm employed, and the degree of completeness of the algorithm. This paper proposes a new, unified, TMS algorithm, called a hybrid TMS (HTMS) which dynamically adapts to the needs of the task.

Three of the main varieties of TMSs are [7, 15]: (1) justification-based TMSs (JTMSs) [11], (2) logic-based TMSs (LTMSs) [16], and (3) assumption-based TMSs (ATMSs) [2]. Some TMSs are optimized to work best with a single context (e.g., a JTMS or LTMS); others function best for multiple contexts (e.g., an ATMS). Applications which do not change contexts often are better served with a JTMS or LTMS. Applications which change contexts often are better served with an ATMS. Unfortunately, many applications, including diagnostic ones [6, 8] only consider a small number of the possible

contexts so the ATMS algorithms waste effort exploring contexts irrelevant to the diagnostic goal. However, there is usually too much context switching to use a single-context TMS effectively. The original descriptions of these TMSs present distinct interfaces to the inference engine using it. Recent authors [7, 18, 19] have argued that the differences in interfaces among TMSs are not fundamental and that a single generic TMS interface suffices. If implementations comply with this interface, then an implementor can substitute one TMS directly for another avoiding the necessity to make early architectural commitments to one TMS over another.

Although the generic TMS interface is a significant advance, it is primarily a cosmetic one as it does not provide a corresponding unified algorithm. For example, the algorithm presented in [19], which uses the generic TMS interface, is essentially an ATMS algorithm. The underlying observation embodied in the generic TMS interface idea is not that the interfaces are unifiable, but rather that the different varieties of TMSs are better thought of as algorithms with differing time-space tradeoffs. Instead of using designations such as 'JTMS', 'LTMS' or 'ATMS' we should all along be talking about 'JTMS algorithm', 'LTMS algorithm' and 'ATMS algorithm.'

The HTMS algorithm has the familiar TMS interface, but adapts to the task — thus the problem-solver designer can avoid making a too-early commitment to one algorithm over another and the HTMS adapts to the task as problem solving unfolds. The HTMS algorithms can be understood either as an extension of an LTMS algorithm or of an ATMS algorithm. However, it is simpler to start with the ATMS conceptualization. The HTMS algorithm can be viewed as an ATMS algorithm with four important modifications:

- The HTMS only maintains labels with respect to a specified set of contexts — each context characterized by a focus environment. This avoids much of the combinatorial explosion encountered using the ATMS as irrelevant label environments lieing outside of the focus environments are not maintained.

- For every focus environment, the HTMS finds at most one label environment for each TMS node. This eliminates the combinatorial explosion (which occurs in diagnostic tasks) where a node has an exponential number of derivations within the same context.

- The HTMS incorporates a scoring function which guides the HTMS to find the single 'best' label environment for each node. For example, the score of an environment can be the number of assumptions it contains. This scoring function is important in backtracking where we would like to find the smallest nogood supporting the current contradiction. It is also important for best-first search applications. While the ATMS always finds the smallest nogood because it finds all of them, the HTMS can be directed to find small ones. The function could also compute a score based on the probabilities of the assumptions.

- To improve the expressive power it accepts arbitrary clauses, like the LTMS. However, to bound computational cost it uses an incomplete yet efficient local propagation algorithm analogous to Boolean Constraint Propagation (BCP)[7, 15, 16, 22].

- It exploits extensive ATMS-like caching strategies so that the focus environments can be changed at relatively little cost. Changing contexts is almost always much cheaper than that of an LTMS but more expensive than that of an ATMS.

If the inference engine remains within a single context, then the HTMS behaves like a conventional JTMS or LTMS and if the inference engine explores all contexts, then the HTMS behaves like an ATMS. The HTMS enforces neither an LTMS nor an ATMS framework. Instead the inference engine can control the computation of the HTMS to only focus on gathering the most useful information which is relevant to answering queries about the contexts of interest. The result is often greatly improved efficiency (see Table 3).

There are two important costs incurred in using an HTMS. The first cost lies in the inference engine using the TMS. Although the HTMS-algorithm can be used in place of the ATMS-algorithm directly, the result can be unsatisfactory because inference engines using the ATMS either change their current context 'too often' or (almost equivalently) inquire unnecessarily about all contexts — this is simply because context switching in ATMS is truly free and there is no computational motivation for designers to do otherwise. To exploit the HTMS these problem solvers must be redesigned to explicitly utilize a focus environment with respect to which queries are made. Fortunately, many ATMS-based inference engines operate with the notion of a solution

context (called interpretations in qualitative physics; diagnoses in diagnostic systems). The second cost lies within the HTMS itself. The HTMS can outperform the LTMS because it caches much more, but as a consequence it uses more memory and requires a larger working set. When the task becomes large enough the overhead of caching outweighs the advantages and the LTMS performs better than the HTMS simply because the HTMS-based implementation working set exceeds the physical memory of the machine.

The HTMS has been completely implemented and tested on a large suite of examples over a number of years, particularly on diagnostic and qualitative physics tasks. Its design has been refined through extensive performance monitoring and experimentation. The performance of the HTMS on diagnostic tasks is typically exponentially better than the ATMS. A diagnostic task involving $50,000$ assumptions have been solved in less than 15 seconds of CPU time on a Symbolics XL1200, while the same task using an ATMS could probably not be solved in the age of universe. Section 9 presents the specific performance improvements of the HTMS over the ATMS. Using this HTMS the GDE and Sherlock framework can be extended [10] to be very efficient for arbitrary devices consisting of 1000s of components.

Even the original ATMS incorporated simple strategies to control the problem solver using the ATMS [2]. Most of these earlier focusing techniques restricted the introduction of new justifications (e.g., by restricting consumer execution) [5, 14]. However, unless the ATMS itself is also controlled, controlling the inference engine alone is insufficient to obtain good performance. Dressler and Farquhar [12] observe this and propose focusing the ATMS to only compute label environments for certain focus environments. However, we have found that alone is insufficient control for many applications (such as diagnosis) — the HTMS finds only one, high-scoring, label environment for each focus environment. Collins and DeCoste [1] have developed an approach which introduces new intermediate assumptions to prevent label explosions but at the cost of significant extra bookkeeping such that the net efficiency gain is often marginal. Tatar [21] proposes a 2vATMS which maintains a label of focus environments which compute more detailed environments only when requested. The 2vATMS is more in the spirit of the HTMS, maintaining two views simultaneously. By maintaining a set of focus environments under which nodes hold, it becomes much simpler to determine which labels to update when a justification is added. However, the list of

supporting environments is *always* (and usually very much) smaller than the list of focus environments under which a node holds.

# 2 Definition of an ATMS/CMS

A Clause Management System (CMS)[19, 20] is a simple generalization of an ATMS so we adopt this cleaner, more general, framework. Every datum which the inference engine reasons about is considered a propositional symbol. The inference engine can refer to both the positive and negative instances of a symbol. These propositional literals are the CMS *nodes*. Every important derivation made by the inference engine is stated as a set of propositional clauses. Throughout this paper $\mathcal{C}$ refers to the set of clauses which have been communicated to the CMS.

The inference engine designates a subset of the literals to be *assumptions*. A CMS *environment* is a set of assumptions. An environment $E$ is inconsistent (called *nogood*) if $E \cup \mathcal{C}$ is not satisfiable. A node $n$ is said to be true (or hold) in environment $E$ if $n$ can be propositionally derived from $E \cup \mathcal{C}$. A node $n$ is said to be false in environment $E$ if $\neg n$ is true in it. Thus, given a node $n$ and environment $E$, $n$ is either true, false, or unknown in $E$. Unknown corresponds to the situation where adding neither the clause $n$ nor the clause $\neg n$ to $\mathcal{C}$ would make $E$ nogood. A nogood is *minimal* if it contains no others as a subset.

The CMS is incremental, receiving a constant stream of additional nodes, additional assumptions, additional clauses and various queries concerning the environments in which nodes hold. To facilitate answering these queries an ideal CMS algorithm maintains for each node $n$ a set of environments $\{E_1, \ldots, E_k\}$ (called the *label*) having the four properties:

1. [Soundness.] $n$ holds in each $E_i$.
2. [Consistency.] $E_i$ is not nogood.
3. [Completeness.] Every consistent environment $E$ in which $n$ holds is a superset of some $E_i$.
4. [Minimality.] No $E_i$ is a proper subset of any other.

The notation $\langle x, L \rangle$ indicates $L$ is the label for $x$. For example,

$$\langle \neg\beta, \{\{B, C\}, \{D\}\} \rangle,$$

indicates that the literal $\neg\beta$ has label environments $\{B, C\}$ and $\{D\}$.

Given the label data structure CMS algorithms can efficiently answer the query whether a node $n$ holds in environment $E$ by checking whether $E$ is a superset of some $E_i$. The CMS algorithm also maintains a data-base of all minimal nogoods to facilitate maintaining label consistency.

For most applications, completely ensuring all four node properties for all literals and all clauses is computationally impractical for it amounts to computing prime implicates. Therefore, most practical CMS implementations only do simple forward propagation. Simple propagation is correct for Horn clauses and positive literals, but may produce incorrect results otherwise.

## 2.1 Two inefficiencies inherent in the ATMS

ATMS labels can grow exponentially in the number of assumptions. This, in itself, is not necessarily an issue were it not for the observation that for many problem-solving tasks most of the environments in these large labels are irrelevant to the task at hand.

Let us examine a case where this exponential, but irrelevant growth occurs. The following example forces the ATMS to construct labels which grow exponentially in size with the clauses it is supplied ($\mathcal{C}$). The idea is to create a node whose label contains all $n$-bit binary integers. Let $b_i = 0$ and $b_i = 1$ be assumptions for $1 \leq i \leq n$. Let $k_i$ represent "integers of size $i$." The formula schemas (which can be easily converted to their constituent clauses) which achieves this is:

$$\begin{aligned} & k_0, \\ b_i = 0 \wedge b_i = 1 & \rightarrow \bot, \\ b_i = 0 \wedge k_{i-1} & \rightarrow k_i, \\ b_i = 1 \wedge k_{i-1} & \rightarrow k_i. \end{aligned} \tag{1}$$

These clauses produce the following labels for $k_1$, $k_2$ and $k_3$:

$$\langle k_0, \{\{\}\} \rangle$$

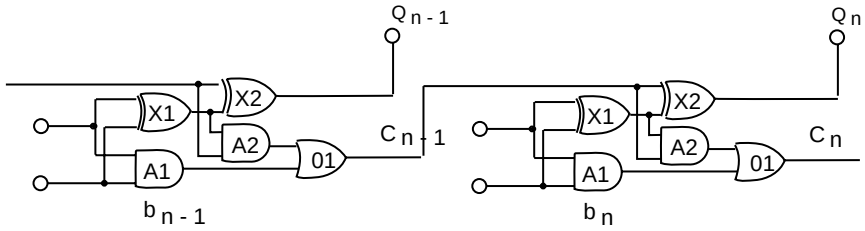



Figure 2: An $n$-bit ripple carry adder.

$$\langle k_1, \{\{b_1 = 0\}, \{b_1 = 1\}\}\rangle$$

$$\langle k_2, \{\{b_2 = 0, b_1 = 0\}, \{b_2 = 0, b_1 = 1\}, \{b_2 = 1, b_1 = 0\}, \{b_2 = 1, b_1 = 1\}\}\rangle$$

Thus, the label of $k_i$ has $2^i$ environments in it. If the inference engine requires the full label for $k_i$, then the exponential work is unavoidable. However, if the inference engine is interested in only a few of the environments, then the ATMS has done exponentially more work than necessary.

The second problem is that labels can grow exponentially even within a single context. The exponential label of the previous example arises because there are an exponential number of distinct maximal contexts. This suggests that a reasonable strategy is to limit label construction to the particular contexts of interest (also suggested in [12]). This is insufficient in practice as is illustrated by modifying the previous example as follows. Remove the clauses requiring that $b_i = 0$ and $b_i = 1$ be inconsistent (i.e., those arising from Formula 1). With this modification no context is inconsistent with any other and there is exactly one maximal context consisting of all the assumptions $b_i = 0/1$. If the inference engine were only interested in determining whether $k_i$ holds in the context containing all the assumptions, then only a single label environment for $k_i$ would be sufficient — the ATMS has done exponentially too much work.

Both of these problems arise extensively in diagnostic tasks [6, 8, 10]. Consider the $n$-bit adder illustrated in Figure 2. Using GDE, each component is modeled by four modes: $G$ (good), $S1$ (output stuck at one), $S0$ (output stuck at zero), or $U$ (unknown mode). Suppose all of the inputs are 0. Just describing this case to GDE causes an exponential amount of work in $n$. Consider $Q_n = 1$ (i.e., the faulty output). The environments in this node's

label will represent all possible combinations of faulty components which produce the faulty output $Q_n = 1$.

For $n = 1$ its label is: $\{S1(B0.X2)\}, \{G(B0.X2)S1(B0.X1)\}$.

For $n = 2$ its label is:

$\{\{S1(B1.X2)\}$

$\{G(B1.X1)G(B1.X2)S1(B0.O1)\}$

$\{G(B0.O1)G(B1.X1)G(B1.X2)S1(B0.A2)\}$

$\{G(B0.O1)G(B1.X1)G(B1.X2)S1(B0.A1)\}$

$\{G(B1.X2)S0(B1.X1)S1(B0.O1)\}$

$\{G(B0.O1)G(B1.X2)S0(B1.X1)S1(B0.A2)\}$

$\{G(B0.O1)G(B1.X2)S0(B1.X1)S1(B0.A1)\}$

$\{G(B0.A2)G(B0.O1)G(B1.X2)S0(B0.A1)S0(B0.X1)S1(B1.X1)\}$

$\{G(B0.O1)G(B1.X2)S0(B0.A1)S0(B0.A2)S1(B1.X1)\}$

$\{G(B1.X2)S0(B0.O1)S1(B1.X1)\}$

$\{G(B0.A2)G(B0.O1)G(B1.X2)S0(B0.A1)S1(B1.X1)\}$

$\{G(B0.A1)G(B0.A2)G(B0.O1)G(B1.X2)S1(B1.X1)\}$

$\{G(B0.A1)G(B0.O1)G(B1.X2)S0(B0.A2)S1(B1.X1)\}\}$.

For $n = 3$, its label has 31 environments.

For $n = 4$, its label has 61 environments.

For larger $n$, label size is roughly $2^{n+2}$.

Suppose we now measure $Q_n = 1$. For any $n$, this circuit has only 5 likely faults which all lie in the final two stages of the adder (a faulty carry in the next to last bit, or a faulty gate contributing to the final output). GDE has performed exponentially too much work.

Suppose we somehow focused the ATMS to only generate labels relevant to diagnoses of interest. This significantly reduces the complexity of the labels in this case, but a significant unnecessary slowdown remains. Suppose we have not yet measured the output of a two stage ripple carry adder. As the inputs are all 0, the carry out must be 0. Suppose we restrict the ATMS to only generate environment labels assuming all components are working correctly. For a 2 stage adder, the carry out of 0 can be determined in two different ways. Intuitively, to obtain a non-zero carry out from the final stage

| $n$ | Gs | Faulted |
|---|---|---|
| 2 | 2 | 2 |
| 3 | 7 | 13 |
| 4 | 15 | 39 |
| 5 | 26 | 80 |
| 6 | 40 | 136 |
| 7 | 57 | 207 |
| 8 | 77 | 293 |

Table 1: Number of superfluous label environments. '$n$' is the number of bits of the adder. 'G' lists the number of superfluous environments for the diagnosis in which all components are unfaulted. 'Faulted' lists the number of superfluous environments in the case where the output is measured to be 1 and the leading 5 diagnoses have been identified.

requires 2 of the 3 inputs to the final stage (two direct inputs and the carry in to be 0). As a result the node $C_1 = 0$ has two possible environments:

$$\{G(B0.A1)G(B0.A2)G(B0.O1)G(B1.A1)G(B1.A2)G(B1.O1)\}$$

$$\{G(B1.A1)G(B1.A2)G(B1.O1)G(B1.X1)\}$$

In general, $C_n = 0$ has $n$ label environments. As $n$ increases an ever increasing number of the nodes have such multiple labels. As Table 1 indicates, the number of superfluous environments thus grows by $n^2$. For larger devices this $n^2$ effect produces a significant slowdown. Moreover, for circuits with very different topologies (e.g., cascades of Or gates feeding into And gates) there will be an exponential number of superfluous environments in labels.

# 3 Definition of an LTMS

Like the CMS, the LTMS is supplied a set clauses $\mathcal{C}$. There is no necessity to differentiate assumptions from other literals. Instead the inference engine

fixes the current context by supplying an initial set of literals $\mathcal{A}$. The task of the LTMS is to label each symbol **T** (true) or **F** (false) depending on whether it or its negation logically follows from $\mathcal{C} \cup \mathcal{A}$. In addition, if $\mathcal{C} \cup \mathcal{A}$ is inconsistent, the LTMS must signal the inference engine with a set of literals $\mathcal{A}' \subset \mathcal{A}$ (i.e., a nogood) such that $\mathcal{A}' \cup \mathcal{C}$ is inconsistent. As problem solving proceeds $\mathcal{C}$ grows monotonically, but $\mathcal{A}$, which specifies the current context, changes non-monotonically as the inference engine focuses on different areas of the problem-solving task.

Full LTMS implementations are slow, therefore most implementation utilize the efficient but incomplete BCP algorithm. To ensure soundness the problem solver can use backtrack search to find labelings which satisfy all the clauses.

## 3.1 Two inefficiencies inherent in single context TMSs

Many of the problems with a single-context TMS algorithms have been extensively discussed elsewhere [2, 15]. However, there are two main ones which concern us here. First, context switching has significant cost. In the worst case, context switching requires checking the label of every node. Often only a fraction of the nodes will be affected, but nevertheless the context switch requires an extensive, and in worst case, global operation.

There is a second, more serious, problem with a single context TMS. When the current context becomes inconsistent, the problem solver must be provided a good reason for why it has become inconsistent (e.g., a small nogood). This information is crucially important to be able to select a next context to work on. If the nogood found is not minimal, the dependency-directed backtracker may move to a new context which is inconsistent for one of the reasons a previous one was — but was not detected because the minimal nogood was not found in the earlier context. Typically, the inconsistencies within a context can be derived in many ways, resulting in different nogoods. The data structures constructed by current single-context JTMS and LTMS algorithms are typically inadequate to find the most constraining nogoods fast. Consider how one finds nogoods with a typical JTMS algorithm. The inconsistency is detected by some contradiction node gaining support (coming in). The typical JTMS implementation records with each node a reason (i.e., a justification or clause) which forced its label. For each

supported contradiction node, these reasons, called supporting justifications in JTMSs, is traversed by treewalking to find a single nogood. Unfortunately, this treewalk can be costly and finds an arbitrary nogood. The particular nogood found is an artifact on of the internal details of the algorithm. Different implementations find very different nogoods. As in many problem-solving tasks most of the contexts are eventually determined to be inconsistent, lack of constraining nogoods is a major detriment to problem-solving performance.

# 4 Specification of an HTMS

Like an ATMS or a LTMS, an HTMS is supplied a monotonically increasing set of clauses $\mathcal{C}$. Like the LTMS and unlike the ATMS there is no need to distinguish assumptions from any other literals and the initial literals $\mathcal{A}$ can change arbitrarily (this is the way the current context is changed for an HTMS or LTMS).

The major interface difference between a conventional single-context TMS algorithm and an HTMS algorithm is that if $\mathcal{C} \cup \mathcal{A}$ is found to be inconsistent, then the HTMS is expected to return a constraining nogood. The inference engine supplies a function $s$ which returns a goodness score for an environment. The HTMS uses this scoring function to identify the most constraining nogoods and informative environments. $s$ should be designed such that environments with lowest score cover the largest fraction of the problem space. This function can be arbitrary so long as it obeys the condition that if $A \subset B$, then $s(A) \leq s(B)$. (This monotonicity stipulation ensures that the HTMS algorithm can continue to discard subsumed label environments.) A very simple scoring function is: $s(E) = |E|$. For diagnostic applications where the problem solver is interested in the most probable diagnoses we use $s(E) = 1 - \prod_{c \in E} p(c)$ where $0 \leq p(c) \leq 1$ is the probability of behavioral mode $c$. $s$ can also encode the preferences for an abduction problem.

The major interface difference between an HTMS and an HTMS is that the label it maintains for nodes is incomplete — for each node it maintains a single, low scoring, *supporting environment* within the focus.

The major performance difference between an HTMS algorithm and a single-context TMS algorithm is its architecture to ensure efficient context switches.

One basic intuition underlies the HTMS algorithms: only compute enough of the ATMS labels that are necessary to find low-scoring nogoods and answer queries. Thus, the HTMS has the functional advantages of the ATMS without most of its computational overhead.

The HTMS can easily simulate a LTMS. The LTMS label for a symbol $x$ is determined as follows. If the HTMS label of $x$ contains an $E_i \subset \mathcal{A}$, then $s$ has the LTMS label **T**, if the HTMS label of $\neg x$ contains $E_i \subset \mathcal{A}$, then $x$ has the LTMS label **F**, otherwise $x$ has the LTMS label **U**.

Consider the first example of Section 2.1. If initially $\mathcal{A} = \{b_1 = 0, b_2 = 0\}$, then the HTMS computes labels:

$$\langle k_0, \{\{\}\}\rangle$$

$$\langle k_1, \{\{b_1 = 0\}\}\rangle$$

$$\langle k_2, \{\{b_2 = 0, b_1 = 0\}\}\rangle$$

At worst each change to $\mathcal{A}$ can add one new environment to a node's label. The size of a label can never grow larger than the number of focus environments explored. This shifts the exponential into the inference engine where it can be better controlled.

In the second example, suppose $\mathcal{A} = \{b_1 = 0, b_1 = 1, b_2 = 0, b_2 = 1\}$ and $p(b_i = 0) = 0.9, p(b_i = 1) = 0.1$ and $s(E) = 1 - \prod_{c \in E} p(c)$. Then, the HTMS computes the labels:

$$\langle k_0, \{\{\}\}\rangle$$

$$\langle k_1, \{\{b_1 = 0\}\}\rangle$$

$$\langle k_2, \{\{b_2 = 0, b_1 = 0\}\}\rangle$$

Even though there may be a large number of possible label environments for a given $\mathcal{A}$, the HTMS adds only the best environment to each label.

# 5 Completeness vs. Efficiency

Most LTMS implementations are based on BCP — an efficient yet incomplete algorithm which has found widespread applicability. BCP is logically complete for Horn clauses, but is not for arbitrary clauses. BCP can be made

complete by adding prime implicates to the input clause set [9], but this is usually too expensive. Analogously, our HTMS implementation is based on BCP.

The HTMS labels obey the conventional ATMS soundness and minimality properties. They also obey a limited consistency property that no label environment may be subsumed by any known nogood (i.e., that has been discovered by HTMS algorithms). For node $n$ with label environments $E_1$, ..., $E_k$ the following properties must hold:

1. [Soundness.] $n$ holds in each $E_i$.
2. [Consistency.] $E_i$ is not *known* nogood.
3. [Minimality.] No $E_i$ is a proper subset of any other.

The HTMS guarantees that if $\mathcal{A} \cup \mathcal{C}$ can be determined to be inconsistent via BCP, then a low scoring nogood is returned. The HTMS guarantees (unless a nogood is found) that the label contains a supporting environment for node $n$ given $\mathcal{A}$, if one exists. $E$ supports node $n$ in $\mathcal{A}$ if:

1. $E \subset \mathcal{A}$,
2. $E \cup \mathcal{C} \vdash n$,
3. For every $E'$ obeying the previous two properties, $s(E) \leq s(E')$.

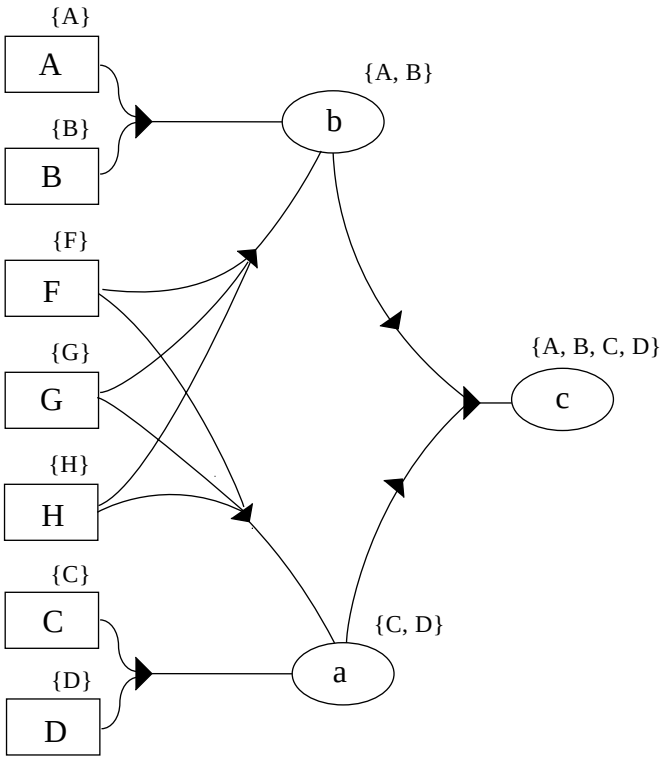
In case where a node has multiple supporting environments, the HTMS consistently chooses one.

Unfortunately, a local propagation algorithm cannot always compute the smallest possible supporting environments even when the clauses are Horn. Consider the following clauses with $s(E) = |E|$:

$$A, B \rightarrow a \quad (1)$$
$$C, D \rightarrow b \quad (2)$$
$$F, G, H \rightarrow a \quad (3)$$
$$F, G, H \rightarrow b \quad (4)$$
$$a, b \rightarrow c \quad (5)$$

Given $\mathcal{A} = \{A, B, C, D, F, G, H\}$ the best environment for $a$ is $\{A, B\}$ and the best environment for $b$ is $\{C, D\}$. If the clauses are supplied in the

Figure 3: Local propagation is suboptimal. The labels of $b$ and $c$ are minimal, but the label for $c$ is not. The minimal label of $c$ should be $\{F, G, H\}$.




order indicated then the algorithm produces the supporting environment $\{A, B, C, D\}$ for $c$ while $\{F, G, H\}$ is globally optimal (see Figure 3). If clauses 1 and 2 were supplied last, then it would find this global optimal for $c$. Thus, the result of this implementation is dependent on the order in which clauses are supplied.

Extending our algorithm to be complete for general clauses yields a far too inefficient TMS. Extending the algorithm to be complete for Horn clauses is easily doable but significantly increases the computational expense: Run the current HTMS algorithm until quiescence, set a global threshold which is the maximum score of all label environments. Then locally propagate the HTMS label with an ATMS-style algorithm cutting off all environments which score

worse than the threshold (a suggestion made by David McAllester). Clearly this generates globally optimal labels in the previous example (it will generate globally optimal nogoods also).

This added inefficiency is rarely worth it. The HTMS algorithm computes close to optimal labels. Table 2 compares the HTMS algorithm to optimal. The results of the table illustrates that the BCP-based HTMS algorithm is nearly optimal for some of the most common ATMS applications. Given the inefficiency of a completely HTMS algorithm, we will always use the BCP-based algorithm.

# 6 HTMS Algorithms

The core of the HTMS algorithm is a best-first propagator which propagates environments with best scores first. However, for the algorithm to be reasonably efficient we must maintain some important data structures which we discuss first.

By far the most common HTMS internal operation is to check whether a particular environment $E$ lies in in focus, i.e., to check whether $E \subset \mathcal{A}$. Therefore, our implementation stores a count $c(E) = |E - \mathcal{A}|$ with each environment $E$, i.e., the number of out-of-focus assumptions $E$ contains. This count is only maintained for those environments which appear in some node's label (i.e., not for environments created in purely internal HTMS operations) and nogoods. It is rarely necessary to compute this count from the environment's constituent assumptions. As the HTMS only propagates environments within the current focus, all environments added to labels will be within the current focus initially and therefore have a count of 0. Whenever an assumption is added or retracted the counts of the label environments which contain it are incremented and decremented.

Whenever the HTMS updates labels for $\mathcal{A}$ it flags $\mathcal{A}$ as completely processed. This ensures that a context switch to a previously completely analyzed context which has not been affected by subsequent changes to $\mathcal{C}$ is free.

The following algorithms gain a significant speed advantage by recording with each assumption all the environments in which it appears. Likewise each environment has a field which points to all the nodes in whose label it

| | | Average Environment | | | Max environment | | |
|---|---|---|---|---|---|---|---|
| Task | A | Optimal | HTMS | Loss | Optimal | HTMS | Loss |
| A-2 | 40 | 2.8 | 2.8 | 1.0 | 5.5 | 5.5 | 1.0 |
| A-3 | 60 | 3.0 | 3.0 | 1.0 | 6.3 | 6.3 | 1.0 |
| A-4 | 80 | 3.2 | 3.2 | 1.0 | 7.0 | 7.0 | 1.0 |
| A-5 | 100 | 3.4 | 3.4 | 1.0 | 7.0 | 7.0 | 1.0 |
| P-2 | 32 | 2.2 | 2.2 | 1.0 | 3.2 | 3.2 | 1.0 |
| P-5 | 96 | 3.1 | 3.1 | 1.0 | 8.4 | 8.4 | 1.0 |
| EX1 | 19 | 2.23 | 2.29 | 1.03 | 3.5 | 3.8 | 1.09 |
| EX2 | 37 | 2.47 | 2.65 | 1.07 | 3.69 | 4.71 | 1.28 |
| EX3 | 58 | 2.43 | 2.43 | 1.0 | 4.61 | 4.61 | 1.0 |
| EX4 | 34 | 3.05 | 3.91 | 1.28 | 2.40 | 2.61 | 1.09 |
| EX5 | 55 | 2.29 | 2.29 | 1.00 | 2.31 | 2.31 | 1.00 |
| EX7 | 84 | 6.68 | 6.70 | 1.00 | 2.75 | 2.75 | 1.00 |

Table 2: Average globally minimal environment size on a series of problem-solving tasks compared to the minimal environment computed using an HTMS algorithm based on BCP. The set of examples A-$n$ diagnose a set of different sized adders; the set P-$n$ a set of parity circuits. The tasks EXn are derived from qualitative physics reasoning tasks. The table is computed by averaging over all nodes and all contexts representing solutions to the task. 'A' indicates the number of assumptions. The 'Average Environment' indicates the average size of all environments produced for every interpretation constructed in the task. 'Max environment' is the average of the largest environment of each interpretation constructed. 'Optimal' is the result obtained by choosing the best environment from a full ATMS label. 'HTMS' is the figure for the HTMS. 'Loss' is the ratio of the the 'Optimal' and the 'HTMS' columns. It is a crude measure of how suboptimal the HTMS is in practice.

appears.

The HTMS is constantly computing the supporting environments for nodes. The HTMS caches the results of these computations by maintaining a tuple $[H, E]$ with each node $n$ where $E$ (or $\phi$ if none) is a supporting environment for $n$ when $\mathcal{A} = H$. This tuple is constructed and updated lazily. The following algorithm returns the supporting environment for $n$:

ALGORITHM **SUPPORT** $(n)$

1. If the cache $[H, E]$ for node $n$ is nonempty and $H = \mathcal{A}$, then return $E$.
2. Find a lowest scoring in focus environment $E$ (i.e., $c(E) = 0$) in $n$'s label. Set $E$ to $\phi$ if there is no supporting environment.
3. Set the cache to $[\mathcal{A}, E]$.
4. Return $E$.

Whenever an environment is added or removed from the label of a node its cache is updated accordingly.

We can now present the best-first propagation algorithm. For simplicity, we assume all clauses are Horn and all assumptions are positive (this is easily generalized). Our implementation operates by maintaining the invariant that if the current context (specified by $\mathcal{A}$) is (BCP-) consistent, then every label must contain a supporting environment (if one can be found via BCP) for every node. When invoked, the HTMS does not relinquish control back to the inference engine until this invariant holds.

The HTMS algorithms maintain a queue $Q$ of justifications/clauses to process. When problem solving begins this $Q$ is empty. Each justification/clause can appear at most once on $Q$ and the score of the environment that $J$ would produce for its consequent node is stored with $J$. (In practice, most justifications/clauses placed on $Q$ are never processed because better environments are found first via other justifications or $\mathcal{A}$ is determined to be inconsistent before processing much of $Q$. Therefore, it is not worth completely constructing the environment and storing it with $J$. Instead it is reconstructed when the best-first propagator actually processes $J$.) $Q$ is ordered by ascending score. The $Q$ is never cleared, it contains justifications

that the HTMS must still examine. If an HTMS operation detects no contradiction, then $Q$ will be empty when control is passed back to the inference engine. If an HTMS operation detects a contraction, then processing is immediately halted and the HTMS returns leaving a non-empty $Q$. This $Q$ will be re-examined on the next HTMS operation.

If $\mathcal{A}$ contains no nogood, then the HTMS processes a new justification $J$ as follows:

**ALGORITHM NEWJUST($J$)**

1. Set the first element of each node cache to $\phi$. Mark all existing focus environments as incomplete. [1]
2. Compute the score of the environment $J$ produces. If there is no such environment, we are done.
3. Mark $J$ with the score, and add $J$ in $Q$.
4. Call **BESTFIRST**

The following algorithm will identify a new nogood if one exists and update the label data structures such that supporting environments can be computed easily if needed:

**ALGORITHM BESTFIRST**

1. Remove the first $J$ from $Q$. If none, then we are done. Mark $\mathcal{A}$ as 'complete'.
2. Clear the score of $J$.
3. Let $E =$ **SUPPORT**($n$) where $n$ is the consequent of $J$.
4. Do a precheck to determine whether the union of the supporting environments of the antecedents of $J$ would be a superset of $E$. If so $J$ can be skipped, go to step 1.

[1] In the actual Common Lisp implementation these operations are accomplished by using time stamps.

5. Compute the union $E'$ of the supporting environments of all the antecedents to $J$. If there is no $E'$, go to Step 1. Set its count to 0. If $n = \bot$, then after removing $E'$ and all of its supersets from node labels immediately return this nogood.

6. If $E'$ is subsumed by some other environment in $n$'s label, go to step 1.

7. Merge $E'$ to the existing label as usual for an ATMS.

8. If $s(E') \geq s(E)$, go to step 1.

9. Make $E'$ the new supporting environment for $n$ in $\mathcal{A}$.

10. Examine each justification $J'$ in which $n$ appears and if each of the other antecedents of $J'$ have current supporting environments then compute the score and insert or update $J'$ in Q. Go to step 1.

Most changes to $\mathcal{A}$ are relatively small and can be viewed as retracting and enabling small sets of assumptions. The following adds a new assumption to $\mathcal{A}$.

ALGORITHM **ADD(A)**

1. $\mathcal{A} = \{A\} \cup \mathcal{A}$.

2. Decrement the count $c(E)$ for every label environment $E$ (appearing in some label) which references $A$.

3. If the count of any nogood becomes 0, then restore $\mathcal{A}$ and all the counts $c(E)$ to their values before **ADD** was called and signal a contradiction.

4. If $\mathcal{A}$ is marked 'complete', then return.

5. For every node $n$ which has a label environment $E$ whose count has just become 0 (obtained by following the HTMS data structure pointers):

   (a) If the cache for $n$ is of the form $[\mathcal{A}, e]$, then do nothing.

   (b) If the cache contains $e$ (most common case), $c(e) = 0$, and $s(E) \geq s(e)$, then do nothing.

(c) Find every $J$ in which $n$ appears as an antecedent.

(d) Compute the score of the environment $J$ produces. If there is no such environment, skip that $J$.

(e) Mark $J$ with the score, and add $J$ in $Q$ (or move it ahead if already on $Q$).

6. Call **BESTFIRST**.

The following algorithm retracts an assumption from $\mathcal{A}$. This algorithm handles the more difficult case where the last HTMS operation produced a nogood.

ALGORITHM **RETRACT(A)**

1. $\mathcal{A} = \mathcal{A} - \{A\}$.
2. Increment the count $c(E)$ of every $E$ which references $A$ in some label.
3. If $\mathcal{A}$ is marked 'complete', do nothing.
4. For every $c(E)$ which has just incremented to 1 do the following. For every node $n$ for which $E$ is the currently cached supporting environment, recompute another supporting environment and if it has one, then insert on $Q$ all justifications for $n$ all of whose other antecedents have supporting environments.
5. Call **BESTFIRST**

The full HTMS implementation contains another innovation to batch interactions. The inference engine can supply a sequence of justifications, retractions, and additions, which are then batched as one operation. In this case, the call to **BESTFIRST** is delayed until a query is made. This is significantly more efficient than executing a sequence of calls to **BESTFIRST** each of which is given a bit more information which changes the ordering produced by the previous call. Of course, this this futile work only can be avoided if there are no interspersed queries of labels or checks for contradictions. The batching tactic is most useful in cases where the inference engine can supply a sequence of justifications, retractions, and additions without

much computational cost. If the inference engine must perform a lot of work before the next TMS operation, then it is unwise to batch operations together because that inference engine work might well be futile.

# 7 Complexity Analysis

In Section 9 we present the experimental results of using these HTMS algorithms on a variety of cases. The HTMS algorithm's worst-case complexity can be determined relatively straightforwardly. Let $a$ be the maximum number of assumptions employed in a task. Let $m$ be the number of literal occurrences in $\mathcal{J}$ (e.g., if there are two justifications one with 2 antecedents and another with 3, then $m = 5$). Let $c$ be the number of contexts explored. The worst-case complexity for BCP is $m$ [15]. The worst-case complexity of ATMS algorithms are $o(ma2^a)$. The worst-case complexity of our HTMS algorithm is $o(mac)$.

A single invocation of **BESTFIRST** can cost at most $o(mac)$. This can be seen as follows. Every iteration of **BESTFIRST** attempts to construct an environment which can be at most size $a$. In the worst case this environment is always added to the label, and the label is in the worst case of size $c$. Because of the best-first nature of the propagation, when a node receives an environment, it will never be changed because all environments produced in later iterations will have equal or worse scores. Hence, **BESTFIRST** iterates at most the number of nodes which is typically much less than $m$ times. However, there is another factor of $ma$ which cannot be ignored. Step 10 of the algorithm checks each justification in which the newly updated node appears. No node can be assigned to more than once per call to **BESTFIRST**, so there can be at most $m$ such checks in one call to **BESTFIRST**. As environments can be of size $a$, the complexity of this operation is $ma$. The worst case complexity is thus $o(mac)$.

The worst-case complexity analysis reveals the hybrid nature of the HTMS. If the number of contexts considered are large, then $c$ may approach $2^a$ and the complexity of the HTMS approaches that of the ATMS. But that is not necessarily of concern if the task itself requires $2^a$. If the number of contexts is small, then the complexity of the HTMS approaches that of the LTMS.

From this crude worst-case complexity analysis one might argue that one

should always use BCP instead of the HTMS. BCP's worst-case performance is $o(mc)$ vs. $o(mac)$. In practice, the factor of $a$ is relatively small and the caching features of the HTMS make it perform better than simple BCP even though the worst case complexity of BCP is better.

# 8 Problem solving with the HTMS

Most of the problem-solving strategies which have been developed for use with an ATMS apply, without change, to the HTMS. For example, the consumer architecture of [4] can be used directly. This consumer architecture already incorporates a focusing strategy to limit the execution of inference engine rules. Thus, converting the consumers and rules of an application from an ATMS to an HTMS is almost trivial.

The HTMS must however always be supplied a focus. Fortunately, many ATMS-based problem solvers either use a focus already (to limit rule execution) or can easily be adapted to do so. The Sherlock diagnostic framework [8] always used a focus in order to generate only the most probable diagnoses. Similarly systems such as QPE [13] which repeatedly construct interpretations to guide problem solving (using the justify-interpret cycle) can be directly adapted. In general, most ATMS-based problem solvers which rely on some form of interpretation construction to find solutions should be adaptable to use an HTMS. The basic strategy is to move interpretation construction to be the first thing that is done instead of the last so that these interpretations focus both the HTMS and the inference engine.

To facilitate this style of problem solving our HTMS implementation includes of a version the ATMS interpretation construction algorithm which upon finding a solution environment (i.e., one that contains no known nogoods), focuses the HTMS on it to check if there might be some nogood within it. Only after no nogoods are found after focusing, does it provide the environment to the inference engine as a solution.

# 9 Experimental results

Table 3 summarizes the results obtained on a large variety of diagnostic (A-$n$, P-$n$) and qualitative physics tasks(EX$n$). The table compares the

| | | Label | | | Nogoods | | | Timing(s) | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Task | A | ATMS | HTMS | Gain | ATMS | HTMS | Gain | ATMS | HTMS | Gain |
| A-2 | 40 | 160 | 67 | 2.4 | 14 | 4 | 3.5 | .5 | .2 | 2.5 |
| A-3 | 60 | 387 | 85 | 4.6 | 41 | 5 | 8.2 | 1.4 | .3 | 4.7 |
| A-4 | 80 | 831 | 98 | 8.5 | 92 | 5 | 18.4 | 4.2 | .3 | 14 |
| A-5 | 100 | 1712 | 111 | 15.4 | 191 | 5 | 38.2 | 12.5 | .3 | 42 |
| A-6 | 120 | 3494 | 124 | 28.2 | 386 | 5 | 77 | 40 | .4 | 100 |
| P-2 | 32 | 131 | 56 | 2.3 | 13 | 2 | 6.5 | .2 | .2 | 1 |
| P-5 | 96 | 23333 | 207 | 113 | 2707 | 3 | 902 | 3389 | .6 | 5650 |
| P-8 | 128 | >67763 | >217 | >293 | >2857 | 4 | >714 | n.a. | .95 | n.a. |
| EX1 | 19 | 1971 | 1086 | 1.8 | 72 | 37 | 1.9 | 1.1 | 1.2 | .92 |
| EX2 | 37 | 2831 | 1931 | 1.5 | 227 | 125 | 1.7 | 1.2 | 1.1 | 1.01 |
| EX3 | 58 | 9915 | 3068 | 2.4 | 1282 | 116 | 11.6 | 3.3 | 3.6 | .92 |
| EX4 | 34 | 1996 | 1329 | 1.6 | 152 | 96 | 1.6 | 59.5 | 17.8 | 3.4 |
| EX5 | 55 | 2008 | 930 | 2.2 | 414 | 64 | 6.7 | 2.3 | 2.0 | 1.2 |
| EX7 | 84 | 32150 | 6937 | 4.6 | 4238 | 373 | 11.4 | 638 | 29.6 | 21.6 |

Table 3: HTMS improvement gains. Using the ATMS the task P-8 never completed and so these figures were taken after 240 hours of Symbolics XL1200 time — the same entire task completes in less than one second with the HTMS.

number of nogoods generated and the cumulative size of all the labels in the ATMS and the HTMS.

Here we see that the HTMS clearly results in an exponential performance improvement in the number of nogoods it finds. The HTMS finds exactly 7 nogoods for all A-$n$ for $n > 2$ while the ATMS finds exponentially many. The missing nogoods are obviously irrelevant to the diagnostic task and represent redundant ATMS work. This so significantly improves the performance of the diagnostic algorithm of GDE and Sherlock [6, 8] that the time spent in the HTMS is a negligible fraction of the total running time. Notice that the improvement in performance appears to grow exponentially in the number of assumptions. For example, in the A-$n$ examples, the addition of 20 assumptions doubles the improvement of the HTMS over the ATMS.

Unlike troubleshooting tasks which require only a few good diagnoses, in many qualitative physics tasks one needs to find all solutions and the entire search space must be explored. Nevertheless, the HTMS still provides a noticeable performance improvement. This suggests that even if the problem-solving task requires finding all solutions, the HTMS still can have an advantage over an ATMS.

The HTMS requires more complex recordkeeping than the ATMS. Thus for simpler tasks, the HTMS does not perform much better than the basic ATMS. The real advantage of the HTMS becomes clear on larger tasks where the exponential buildup in ATMS label datastructures becomes unmanageable. The speedup observed in examples such as A-5, A-6, P-5, EX7 is typical. All large tasks which we have applied the HTMS to yield such performance improvements.

Of course most of the advantages of the HTMS cannot be observed in this table — the problem solver can now handle tasks which were impractical before. Thus, [10] describes the performance of Sherlock on circuits which were impossible to analyze with a conventional ATMS. The largest circuit analyzed there would require 14048 assumptions and with a conventional ATMS diagnosis cannot finish, but with an HTMS the diagnoses can be obtained in about 10 minutes of XL1200 time producing total label size 17000 and 172 nogoods.

# 10 Concluding Observations

For the tasks we have studied, an HTMS-style algorithm is invariably superior to an ATMS-style algorithm. The central reason for this is that an HTMS-style algorithm adapts to the task, limiting TMS operations and data structures to those that are directly relevant to the problem solving task and does this without introducing excessive bookkeeping overhead. The new algorithm is thus one of a small set of algorithms designers should consider in building TMS-based problem solvers.

There is no reason to believe that the HTMS-style algorithm is the best. We are continuing to experiment with different TMS algorithms and different TMS-problem-solver interaction paradigms. One promising possibility is suggested by LTMS algorithms. In this research we did not even experiment with pure LTMS algorithms. This was because some of the first qualitative physics and diagnostic systems were based on LTMS style algorithms. These algorithms turned out to be totally impractical—the number of context switches required for these tasks were too excessive. However, just as an HTMS's design is based on limiting the caching capabilities of an ATMS, one can imagine extending an LTMS with limited caching capabilities. Various such schemes have been proposed but no such scheme has ever been completely developed. In the interest of developing even faster problem solvers, this may be a promising direction to pursue.

# 11 Acknowledgments

I thank Daniel Bobrow, Oskar Dressler, Ken Forbus, David McAllester Olivier Raiman, and Brian C. Williams for giving insightful comments and useful feedback about the ideas in this paper. Nora Boettcher and Barbara Hatton prepared the figures.

# References

[1] Collins, J.W. and D. DeCoste, CATMS: An ATMS which avoids label explosions, *Proceedings AAAI-91*, Anaheim, CA (1991) 281–287.

[2] de Kleer, J., An assumption-based truth maintenance system, *Artificial Intelligence* **28** (1986) 127–162. Also in *Readings in NonMonotonic Reasoning*, edited by Matthew L. Ginsberg, (Morgan Kaufmann, 1987), 280–297.[14]

[3] de Kleer, J. and Williams, B.C., Back to backtracking: Controlling the ATMS, *Proceedings AAAI-86*, Philadelpha, PA (1986), 910-917.

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

[5] de Kleer, J. and Williams, B.C., Back to backtracking: Controlling the ATMS, *Proceedings AAAI-86* Philadelphia, PA (August 1986), 910–917.

[6] de Kleer, J. and Williams, B.C., Diagnosing multiple faults, *Artificial Intelligence* **32** (1987) 97-130. Also in *Readings in NonMonotonic Reasoning*, edited by Matthew L. Ginsberg, (Morgan Kaufmann, 1987), 372–388.

[7] de Kleer, J., Forbus, K., McAllester, D., Tutorial notes on truth maintenance systems, IJCAI-89, Detroit, MA, 1989.

[8] de Kleer, J. and Williams, B.C., Diagnosis with behavioral modes, in: *Proceedings IJCAI-89*, Detroit, MI (1989) 104–109. Also in Readings in Model-Based Diagnosis, edited W. Hamscher, J. de Kleer and L. Console, (Morgan Kaufmann, 1992).

[9] de Kleer, J., Exploiting locality in a TMS, AAAI-90, Boston, MA (1990) 254–271.

[10] de Kleer, J., Focusing on probable diagnoses, in: *Proceedings AAAI-91*, Anaheim, CA (1991) 842–848. Also in Readings in Model-Based Diagnosis, edited W. Hamscher, J. de Kleer and L. Console, (Morgan Kaufmann, 1992).

[11] Doyle, J., A truth maintenance system, *Artificial Intelligence* **12** (1979) 231–272.

[12] Dressler, O., and Farquhar, A., Putting the problem solver back in the driver's seat: contextual control of the ATMS, in *Truth Maintenance Systems*, edited by J.P. Martins and M. Reinfrank, (Springer, 1990) 1–16.

[13] Forbus, K. The qualitative process engine, University of Illinois Technical Report No. UIUCDCS-R-86-1288, December, 1986. Also in *Readings in Qualitative Reasoning About Physical Systems*, edited by Daniel S. Weld and Johan de Kleer (Morgan Kaufmann, 1990), 220–235.

[14] Forbus, K.D. and de Kleer, J., Focusing the ATMS, *Proceedings of the National Conference on Artificial Intelligence*, Saint Paul, MN (August 1988), 193–198.

[15] Forbus, K.D. and de Kleer, J., *Building Problem Solvers*, MIT Press, 1993.

[16] McAllester, D., An outlook on truth maintenance, Artificial Intelligence Laboratory, AIM-551, Cambridge: M.I.T., 1980.

[17] McAllester, D., A widely used truth maintenance system, unpublished, 1985.

[18] McAllester, D., Truth Maintenance, in: *Proceedings AAAI-90* Boston, MA (1990) 1109–1116.

[19] McDermott, D., A general framework for reason maintenance, *Artificial Intelligence* **50** (1991) 289–329.

[20] Reiter, R. and de Kleer, J., Foundations of assumption-based truth maintenance systems: Preliminary report, *Proceedings of the National Conference on Artificial Intelligence*, Seattle, WA (July, 1987), 183–188.

[21] Tatar, M.M., Combining the lazy label evaluations with focusing techniques in an ATMS, in: *Proceedings ECAI-94*, Amsterdam, The Netherlands (1994).

[22] Zabih, R. and McAllester, D., A Rearrangement Search Strategy for Determining Propositional Satisfiability, *Proceedings of the National Conference on Artificial Intelligence*, Saint Paul, MN (August 1988), 155–160.