Exploiting locality in a TMS

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

Abstract

This paper presents a new approach for exploiting Truth Maintenance Systems(TMSs) which makes them simpler to use without necessarily incurring a substantial performance penalty. The basic intuition behind this approach is to convey the locality of the knowledge representation of the problem solver to the TMS. The TMS then uses this locality information to control and restrict its inferences. The new TMSs accept arbitrary propositional formulae as input and use general Boolean Constraint Propagation(BCP) to answer queries about whether a particular literal follows from the formulae. Our TMS exploits the observation that if the set of propositional formulae are converted to their prime implicates, then BCP is both efficient and logically complete. This observation allows the problem solver to influence the degree of completeness of the TMS by controlling how many implicates are constructed. This control is exerted by using the locality in the original task to guide which combinations of formulae should be reduced to their prime implicates. This approach has been implemented and tested both within Assumption-Based Truth Maintenance Systems and Logic-Based Truth Maintenance Systems.

1 Introduction

This paper presents a new practical approach for exploiting Truth Maintenance Systems which makes them simpler to use without necessarily incurring a substantial performance penalty. The basic intuition behind this new approach is to convey the locality of the knowledge representation of the problem solver to the TMS. Many AI problem solvers, particularly those which reason about the physical world, are inherently local — each constituent of the problem (e.g., a process such as flowing, a component such as a pipe, etc.) has a fixed behavioral model. Much of the reasoning can be viewed as propagation: whenever some new signal is inferred to be present the models of the components on which it impinges are consulted to see whether further inferences are possible from it. Many of these AI problem solvers either exploit TMSs to do much of this propagation, or use TMSs to represent the results of propagations. Although widely used, anyone who has used these strategies can attest that current TMSs manifest some surprising logical incompleteness when used in this way. These blind spots result from the fact that locality present in the original model is often completely lost within the TMS.

The TMS framework we present is fully expressive accepting arbitrary propositional formulae as input. Provided with advice from the overall problem solver it is, if needed, logically complete. Propositional satisfiability is NP-complete, but nevertheless often much of the cost of logical completeness can be avoided by exploiting locality. For example, conjunctions of formulae in the model library can be precompiled into their prime implicates to reduce run-time cost. Also, the TMS uses locality information at run-time to determine which combinations of formulae are worth analyzing.

We have implemented our framework and used it with both Assumption-Based Truth Maintanence Systems (ATMSs) [2; 7] and Logic-Based Truth Maintenance Systems (LTMSs) [2; 19; 20; 21]. We have used it to compile models from constraints, confluences, order-of-magnitude reasoning axioms and processes. A longer paper [10] explores the role of this framework in qualitative physics in more detail.

1.1 Encoding models as formulae

Most problem solvers wish to represent arbitrary propositional formulae many of which derive from local constituents of the problem (e.g., component or process models). However, most TMSs lack the expressive power to represent arbitrary formulae. Therefore, one is typically forced to encode the propositional formulae in terms the TMS accepts. For example, [4] provides a variety of ways of encoding propositional formulae for the Assumption-Based Truth Maintenance Systems (ATMSs). Techniques like these are widely used in QPE [13; 14]. Unfortunately, these encodings tend to be extremely cumbersome. The TMSs which accept arbitrary clauses (such as LTMSs) seem to be more powerful because any propositional formula can easily be converted into an equivalent set of clauses by putting it into conjunctive normal form CNF[1].

Unfortunately, complete LTMSs based on clauses are rarely used because they are too inefficient. Instead, all common LTMS implementations use Boolean Constraint Propagation(BCP)[2; 19; 20; 21] on clauses. BCP is a sound, incomplete, but efficient inference procedure. BCP is inherently local considering only one propositional formula (i.e., boolean constraint) at a time. This locality is the source of both its incompleteness and efficiency. Unfortunately, converting a formula to its CNF clauses loses the locality of the original formula. Consider the formula:

$$(x \Rightarrow (y \vee z)) \wedge (x \vee y \vee z) \qquad (1)$$

If y were false, then considering this formula alone, in isolation, we can infer z must be true (this can be seen by the fact that if z were false, the first conjunct $x \Rightarrow (y \vee z)$ requires x to be false, and the second conjunct $(x \vee y \vee z)$ requires x to be true). However, this information is lost in converting the formula to its two CNF clauses:

$$\neg x \vee y \vee z, \quad x \vee y \vee z.$$

Neither of these two clauses can, individually, be used to infer z from $\neg y$. Taken together, of course, they can.

Consider QPE as an example [13; 14]. QPE encodes every qualitative process model as a set of formulae which are eventually encoded as a set of ATMS horn clauses. Within QPE, this set of horn clauses represents a fixed local module, but within the ATMS each clause is treated independently. As some of the formulae cannot be converted to purely horn clauses, the basic ATMS algorithms are incomplete with respect to the original formulae, and the ATMS is therefore incapable of making some simple local inferences which follow from the model alone. QPE deals with this difficulty by adding more clauses (than conversion to CNF would indicate) so that the basic ATMS algorithms can make more inferences than they otherwise would. Part of our proposal is that the set of formulae representing a model be conveyed to the TMS as a single module and the TMS use a complete inference procedure locally on modules. As a result we achieve the kind of functionality that is desired, without incurring substantial performance degradation and without burdening QPE with needless encoding details. This process can be made efficient by recognizing that each model type instantiates the same set of formulae and therefore most of the work can be done at compile time once per model type.

Conceptually, the new TMSs are supplied a set of arbitrary propositional formulae and use general BCP to answer queries whether a particular literal follows from the formulae. BCP is usually applied to clauses but can be applied to arbitrary formulae as well. As input the TMS can accept new propositional formulae to define a module, conjoin two existing modules, or accept a new formula to be conjoined with an existing module. Locally, within each module, the TMS is logically complete. As a consequence, the problem solver can dynamically control the trade-off between efficiency and completeness — if completeness is required, all the modules are conjoined, if efficiency is is required, each formula is treated as an individual module. Later in this paper we present a number of techniques to guide the order in which modules should be conjoined in order to minimize computational cost.

Consider the example of two pipes in series (Fig. 1). Each pipe is modeled by the qualitative equation (or confluence, see [3] for precise definitions) $[dP_l]-[dP_r] = [dQ]$ where P_l is the pressure on the left, P_r is the pressure on the right and Q is the flow from left to right. ($[dx]$ is the qualitative (+, 0, −) value of $\frac{dx}{dt}$). Thus, the attached pipes can be completely modeled by three confluences:

$$[dP_A] - [dP_B] = [dQ_{AB}], \qquad (2)$$

$$[dP_B] - [dP_C] = [dQ_{BC}], \qquad (3)$$

$$[dQ_{AB}] = [dQ_{BC}]. \qquad (4)$$

Suppose we know that the pressure is rising at A (i.e, $[dP_A] = [+]$) and the pressure is fixed at C (i.e., $[dP_C] = [0]$). Considering each component or confluence individually we cannot infer anything about the flows. We only know one of the three variables in confluences (2) and (3), and none of the variables in confluence (4). Therefore, none of the confluences, individually, can be used to infer a new variable value. The only way to determine the behavior is to somehow solve the confluences — but that requires global reasoning over the confluences.

If the individual qualitative equations are converted to their propositional equivalents for a TMS (as many qualitative physics systems do), then $[dP_B]$, $[dQ_{AB}]$ and $[dQ_{BC}]$ remain unknown due to the incompleteness of most TMS's. However, in our TMS if the formulae representing the individual components are merged then $[dQ_{AB}] = [dQ_{BC}] = [+]$ is inferred. As such component combinations reoccur in many systems, this combining can be done once in the model library. To compile this combination, our TMS merges the propositional encoding of the confluences but without the specific inputs ($[dP_A] = [+]$ and $[dP_C] = [0]$). The result is identical to the propositional encoding of the confluence:

$$[dP_A] - [dP_C] = [dQ_{AB}] = [dQ_{BC}].$$

After compiling this combination, and applying the inputs our TMS infers that $[dQ_{AB}] = [dQ_{BC}] = [+]$ far more efficiently than before (i.e., in one step).

A device can always be analyzed by first compiling it without knowledge of any input or outputs. However, compiling a full device model is expensive — it is only useful if we expect to put it in the model library or need to consider many input value combinations. When analyzing a device our TMS does not force the problem-solver to decide whether or not to compile the device beforehand. Our TMS lazily compiles the propositional formulae it is supplied — it only compiles enough to answer queries for the givens it is supplied. When the givens are changed the TMS, if necessary, incrementally compiles more pieces of the device to answer the query. If all possible givens and queries are applied, then the compiled result will be the same as having compiled the full device beforehand.

After developing our approach Section 8 expands on these observations and analyzes its relationship with the qualitative resolution rule [11; 12].

2 BCP on formulae and clauses

As our approach draws on the intuitions underlying BCP, we give a synopsis of it here. (Note that BCP

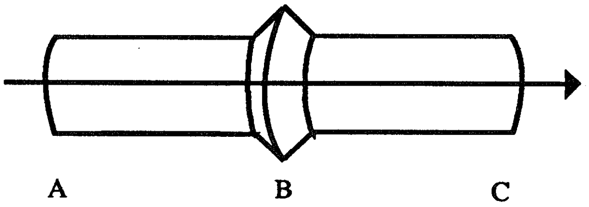

Figure 1: Assembling the qualitative models of the two joined pipes is equivalent to merging the formulae modeling the two pipes.

achieves similar results to unit resolution.) BCP operates on a set of propositional formulae (not just clauses) $\mathcal{F}$ in terms of propositional symbols $\mathcal{S}$. A formula is defined in the usual way with the connectives $\neg, \Rightarrow, \equiv, \vee, \wedge$ and *oneof*. (*oneof* is a useful connective requiring that exactly one of its arguments be true.) For the purposes of this paper a clause is a disjunction of literals with no literal repeated and not containing complementary literals.

BCP labels every symbol **T** (i.e., true), **F** (i.e., false) or **U** (i.e., unknown). BCP is provided an initial set of assumption literals $\mathcal{A}$; if $x \in \mathcal{A}$, then x is labeled **T**, and if $\neg x \in \mathcal{A}$, then x is labeled **F**. $\mathcal{A}$ may not contain complementary literals. All remaining symbols are initially labeled **U**. The reason for distinguishing $\mathcal{A}$ from $\mathcal{F}$ is that $\mathcal{F}$ is guaranteed to grow monotonically while assumptions may be added and removed from $\mathcal{A}$ at any time.

BCP operates by relabeling symbols from **U** to **T** or **F** as it discovers that these symbols logically follow from $\mathcal{F} \cup \mathcal{A}$. A labeling which does not label any symbol **U** is *complete*. Conversely a labeling which labels some symbols **U** is *partial*. A *completion* of a partial labeling is one which relabels all the **U** symbols **T** or **F**. Given any labeling each BCP constraint (in the BCP literature propositional formulae are called constraints) is in one of 4 possible states.

- The labeling *satisfies* the constraint: for every completion of the current labeling the constraint is true. For example, labeling x **T** satisfies the constraint $x \vee y$.
- The labeling *violates* the constraint: there is no completion of the current labeling which satisfies the constraint. Consider two examples: (1) if the constraint is $x \vee y$ and both x and y are labeled **F**, then the constraint is violated, and (2) if the constraint is $(x \vee y) \wedge (x \vee \neg y)$ and x is labeled **F**, then there is no way to satisfy the constraint.
- A constraint *forces* a symbol's label if in every completion of the current labeling which makes the constraint true that symbol is always labeled **T** or always **F**. There may be multiple such symbols. For example, if x is labeled **T**, then the constraint $x \equiv (y \wedge z)$ forces y and z to be labeled **T**. Consider the example from the introduction: $(x \Rightarrow (y \vee z)) \wedge (x \vee y \vee z)$. If y is labeled **F**, then the label of z is forced to be **T**.
- Otherwise a constraint is *open*.

BCP processes the constraints one at a time monotonically expanding the current labeling. The behavior of BCP depends on the condition the constraint is in:

- If the current labeling satisfies the constraint, then the constraint is marked as satisfied and is no longer considered.
- If the current labeling violates the constraint, then a global contradiction is signaled.
- If the current labeling forces the label of some other symbol, then that symbol is labeled and all unsatisfied and unviolated constraints mentioning that symbol are scheduled for reconsideration. If the current constraint is now satisfied it is so marked.
- Otherwise the constraint remains open and BCP reconsiders it when some (other) symbol it references is labeled **T** or **F**.

If this BCP is applied purely to clauses, then the resulting behavior is identical to the clausal BCP dicussed in the LTMS literature.

If the constraints are arbitrary formulae, then determining whether a constraint forces a symbol label is complex to implement and computational expensive execute. However, if the constraints are clauses, then BCP can be implemented simply and efficiently. In particular, we store a count with each clause indicating the number of symbols which are labeled **U** or whose label satisfies the clause. For example, given the clause $x \vee \neg y$ where x is labeled **U** and y is labeled **T**, the count for the clause is 1. Whenever this counter is reduced to 1, then the clause forces the label of a single remaining symbol (i.e., in this case x is forced to **T**). If the count is reduced to 0, then the clause is violated and a contradiction is signaled. As a consequence of this encoding, BCP on clauses can be implemented simply by following pointers and decrementing counters. Conversely, the process of removing an assumption from $\mathcal{A}$ can be efficiently implemented by following pointers and *incrementing* counters. (See [7] for details.) BCP on clauses is equivalent to the circuit value problem and therefore is P-complete (see also [15]). Its worse case complexity is the number of literals in the clauses.

BCP is logically incomplete in that it sometimes fails to label a symbol **T** or **F** when it should. For example, consider the two clauses from the introduction:

$$\neg x \vee y \vee z, \quad x \vee y \vee z.$$

If y is labeled **F**, then BCP on the clauses does not label z **T**. (Note that BCP is also logically incomplete in that it sometimes fails to detect contradictions.)

3 Compiling into prime implicates

The previous example (the encoding of formula (1)) shows that running BCP on the original formulae is usually not the same as running BCP on the clauses produced by converting the formulae to CNF. (BCP on the original formulae is usually much stronger or, at worst, equivalent.) Hence, we cannot directly use the efficient BCP algorithms that have been developed for clauses for arbitrary formulae and no correspondingly efficient BCP algorithm is known. This section shows that if each individual formula is encoded by its prime implicates[16; 17; 22], then BCP on the resulting clauses is equivalent to running BCP on the original formulae.

We use the following definitions. Clause A is subsumed by clause B if all the literals of B appear in A. Therefore if A subsumes B, then B is true wherever A is. An implicate of a set of propositional formulae $\mathcal{F}$ is a clause entailed by $\mathcal{F}$ not containing complementary literals. A prime implicate of a set of formulae $\mathcal{F}$ is an implicate of $\mathcal{F}$ no proper subclause of which is an implicate of $\mathcal{F}$.

Consider the simple example of the introduction. Using the conventional conversion to CNF the formula,

$$(x \Rightarrow (y \vee z)) \wedge (x \vee y \vee z), \qquad (5)$$

is equivalent to the conjunction of the clauses,

$$\neg x \vee y \vee z, \quad x \vee y \vee z.$$

However, there is only one prime implicate,

$$y \vee z.$$

This example illustrates that there may be fewer prime implicates of a formula than the conjuncts in the CNF of a formula. Unfortunately, the reverse is usually the case. Consider the clause set:

$$\neg a \vee b, \quad \neg c \vee d, \quad \neg c \vee e, \quad \neg b \vee \neg d \vee \neg e.$$

In this case, these 4 are all prime implicates, but there are 3 more (for a total of 7):

$$\neg a \vee \neg d \vee \neg e, \quad \neg b \vee \neg c, \quad \neg a \vee \neg c.$$

There are a variety of different algorithms for computing prime implicates (see [6; 7; 9; 16; 22; 23]). Stripped of all the efficiency refinements discussed in the next section, our basic approach is to use a variation of the consensus method to compute prime implicates. First, the formula is converted into CNF to produce an initial set of clauses. Then we repeatedly take two clauses with exactly one pair of complementary literals and construct a resulting clause with both those literals removed. All clauses subsumed by others are removed. This process continues until no new unsubsumed clause is producible.

Using the preceeding definitions, the following theorems are key to an efficient implementation of BCP on constraints:

Theorem 1 *Given a set of clauses $\mathcal{I}$ which are the set of prime implicates of some set of propositional formulae and a set of assumptions $\mathcal{A}$, then if $\mathcal{I} \cup \mathcal{A}$ is inconsistent, then BCP will detect a violation.*

Proof. If $\mathcal{I} \cup \mathcal{A}$ is inconsistent, then there must be a clause,

$$\neg A_1 \vee \cdots \vee \neg A_n,$$

entailed by $\mathcal{I}$ where $A_i \in \mathcal{A}$. This clause is an implicate of $\mathcal{I}$ and thus subsumed by some clause S of $\mathcal{I}$. As all the $\neg A_i$ are labeled **F**, BCP will detect that clause S is violated. □

Theorem 2 *Given a set of clauses $\mathcal{I}$ which are the set of prime implicates of some set of propositional formulae and a set of assumptions $\mathcal{A}$ such that $\mathcal{A} \cup \mathcal{I}$ is consistent, then BCP computes the correct label for every node.*

Proofs for the remaining theorems can be found in [10].

Theorem 3 *Let $\mathcal{A}$ be a set of literals, $\mathcal{F}$ a set of propositional formulae and $\mathcal{I}$ is the union of the prime implicates of each of the formulae of $\mathcal{F}$ individually. If BCP on $\mathcal{A} \cup \mathcal{F}$ does not detect any violations, then BCP on $\mathcal{A} \cup \mathcal{F}$ produces the same symbol labels as BCP on $\mathcal{A} \cup \mathcal{I}$.*

Theorem 4 *Let $\mathcal{A}$ be a set of literals, $\mathcal{F}$ a set of propositional formulae, and $\mathcal{I}$ is the union of the prime implicates of each of the formulae of $\mathcal{F}$ individually. BCP on $\mathcal{A} \cup \mathcal{F}$ detects a violation exactly when BCP on $\mathcal{I} \cup \mathcal{F}$ detects a violation.*

The first two theorems tell us that we can make BCP complete if we need to. The second two theorems tell us that running BCP on the prime implicates of the individual formulae is the same as running BCP on the formulae. Thus, we can exploit the efficient implementations of clausal BCP.

Note that the prime implicates, by themselves, do not solve the task — they represent a family of solutions each characterized by a distinct assumption set $\mathcal{A}$. Computing the prime implicates is analogous to compiling a propositional formula (or set of them) so that it is easy to compute the resulting solution once some input, i.e., $\mathcal{A}$ is provided.

Fig. 2 illustrates some of the options engendered by the theorems. Although replacing the entire set of formulae with their equivalent set of prime implicates allows BCP to be logically complete, the required set of prime implicates can be extremely large. This large set is both difficult to construct and, its very size makes it hard for BCP to work on. Therefore it is usually impractical to exploit this strategy directly.

4 Basic LTMS transactions

Our basic formula LTMS permits the following transactions:

`(add-formula formula)`: This adds an individual formula to the TMS. Section 2 outlines the allowed connectives.

`(add-assumption symbol label)`: This labels the symbol **T** or **F**. This retracts any previous `add-assumption` for this symbol.

`(retract-assumption symbol)`: This removes the initial label for the symbol. Note that the symbol will retain a non-**U** if it follows from $\mathcal{A} \cup \mathcal{F}$.

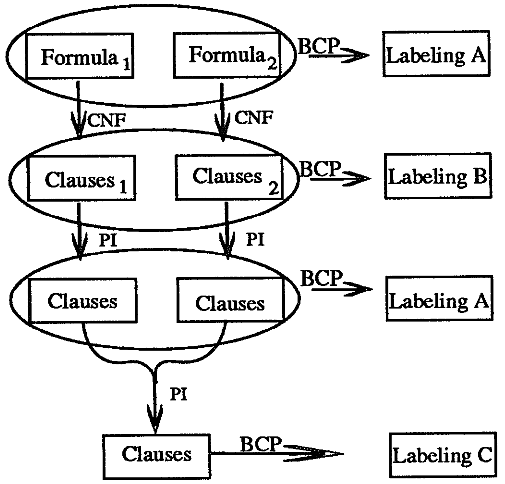

Figure 2: This figure illustrates the different ways BCP can be used. BCP on arbitrary formulae (expensive) produces labeling A. If the formulae encoded as their CNF clauses, then an efficient clausal BCP produces an (unfortunately weaker) labeling B. If formulae are individually converted into their prime implicates, then the efficient clausal BCP finds the same labeling A as the inefficient formula BCP on the original constraints. Finally, if the prime implicates of all the formulae are constructed, then clausal BCP is logically complete.

`(label? symbol)`: This returns the label for the symbol.

`(inconsistent?)`: Tests whether $\mathcal{A} \cup \mathcal{F}$ is inconsistent.

5 Lazy compilation

Using the results of the section 3 a complete formula LTMS algorithm can be implemented as follows. The algorithm always maintains the set of prime implicates $\mathcal{F}'$ of $\mathcal{F}$. Whenever a new formula f is added to $\mathcal{F}$, $\mathcal{F}'$ is updated by computing the prime implicates of the old $\mathcal{F}'$ and the new formula f thus avoiding recomputing the same prime implicates from scratch each time. Whenever $\mathcal{A}$ is changed, we execute the usual clausal BCP algorithm on $\mathcal{F}'$. Therefore, the major cost is paid up front when a formula is added to $\mathcal{F}$ and not when $\mathcal{A}$ is changed.

The algorithm just sketched out is needlessly expensive. In the next few sections we present techniques which substantially improve its performance in most circumstances. The basic intuition behind these techniques to to delay the construction of additional implicates until absolutely necessary because constructing and storing implicates is the most expensive aspect of our framework. In all these techniques $\mathcal{F}'$ is no longer a set of prime implicates of $\mathcal{F}$, but simply a set of implicates of $\mathcal{F}$ none of which is subsumed by another.

Experience has shown that problem solvers often supply a sequence of formulae without making any intervening queries. Thus, calls to **`add-formula`**, **`add-assumption`** and **`retract-assumption`** are not be acted upon until an actual query is made. When the problem solver requests a label of a symbol the following steps are performed:

1. Update $\mathcal{A}$.
2. Perform clausal BCP.
3. If BCP supplies a label for the symbol, we are done.
4. Take the first queued formula f. If there isn't one, then go to step 7.
5. Add each of its prime implicates to the LTMS clause set $\mathcal{F}'$ and mark each such clause as unprocessed.
6. Go to step 2
7. Consider all unprocessed clauses u. Remove all other clauses subsumed by u from $\mathcal{F}'$. Mark u as processed.
8. For two clauses u and v in $\mathcal{F}'$ which contain exactly one complementary pair of literals, which have not been resolved before, compute the consensus of u and v, add it to $\mathcal{F}'$, and remove the clauses it subsumes from $\mathcal{F}'$.
9. If no new clause was added in the previous step and no formulae additions are queued, the the label is **U**.
10. Go to 2.

Although in the naive algorithm which initially computes all prime implicates, retracting an assumption is very simple, in this algorithm retracting an assumption can later provoke further implicate construction because some symbol might lose its **T**/**F** label. Checking for inconsistency is nearly identical. If at any point clausal BCP reports an inconsistency, then no further processing is necessary.

The basic intuition behind this algorithm is to run the efficient clausal BCP before constructing new implicates. This is analogous to the familiar unit-preference strategy except that the initial unit clauses (i.e, $\mathcal{A}$) can be retracted.

The preceeding algorithm still computes far too many implicates than is necessary for a particular $\mathcal{A}$. We can exploit BCP labels to further delay the construction of new implicates. In particular, computing the consensus of two clauses where the complementary literals are labeled **T**/**F** cannot provide new information (for the current $\mathcal{A}$). Therefore, we can add this restriction to the algorithm. This yields substantial performance improvements if not all possible $\mathcal{A}$ are explored.

Many efficiency improvements are possible on this basic scheme. An important one results from distinguishing between the two different reasons a symbol can be labeled **T**/**F**: (1) a symbol can be labeled because it follows via BCP from $\mathcal{F}$ alone (we call this a *fixed* label), or (2) a symbol can be labeled because it follows via BCP from $\mathcal{F} \cup \mathcal{A}$ but not from $\mathcal{F}$ alone (we call this a *variable* label). The important difference is that the fixed labels cannot change as $\mathcal{A}$ changes. If a fixed literal label satisfies a clause, then that clause is removed from $\mathcal{F}'$. If a variable literal label satisfies a clause, then the clause is temporarily set aside. If a fixed literal label violates a clause, then the literal is removed from the clause. These results happen anyway as a consequence of constructing implicates. However, BCP achieves them earlier and more efficiently. Note that as variable labels change, clauses which were previously set aside need to be reanalyzed to compute further implicates.

6 Pre-compiling formulae

Many AI problem solvers operate with a knowledge base or component library. Given a particular task, pieces of this knowledge base are instantiated as needed. For example, in Qualitative Process Theory most processes are instantiated with the same fixed set of formulae (but with different symbols). Hence, the schemas for the prime implicates for each model in the library can be constructed a priori, and many implicate constructions can be thus avoided run time.

A single propositional formula may yield a very large number of prime implicates. If some of the symbols of a formula are internal (i.e., appear only in the formula, are guaranteed never to be referenced by any new input formula and are of no further interest to the problem solver), then all the clauses mentioning that symbol can be discarded without affecting the functionality of the TMS. As a result BCP need not stumble over these needless clauses.

The basic formula LTMS transactions which support this insight are:

(compile-formula schema internal-symbols): Used at *compile* time. This converts the formula schema into a set of prime implicate schemas. This is designed to be used when constructing the knowledge base or the model library. **internal-symbols** is a set of internal symbols which are guaranteed not to appear again. Therefore, after computing prime implicates, all clauses mentioning internal symbols are discarded.

(load-formula schema): This takes the prime implicate schemas and communicates them to the TMS.

7 Exercising problem solver control

In many cases, even lazily constructing sufficient implicates of $\mathcal{F}'$ to ensure completeness for the given $\mathcal{A}$ is too costly. In this circumstance the problem solver provides external guidance to control which prime implicates should be constructed and to choose when to give up completeness.

One way to limit the computational cost of the algorithm is instead of running the algorithm on the entire set of formulae, only apply the algorithm to subsets of the formulae. This locality is captured by the notion of *module*. A module is a set of formulae and the LTMS data base consists of a set of modules. The algorithm is restricted to perform subsumption tests and consensus constructions only within modules. But the clausal BCP is run across all clauses of all modules. The problem solver is provided an additional interface to control when to modules are to be merged. This requires the following additional transactions:

load-formula* and **add-formula***: These create modules initially containing only their formula argument.

(merge-modules module1 module2): This tells the TMS to conjoin the two modules, by computing the necessary implicates of the combination.

(internal symbol): Used at run time. This informs the TMS that the symbol is internal. If all occurrences of this symbol appear in the same module, than all clauses mentioning this symbol can be discarded. This greatly reduces the number of clauses the TMS needs to consider.

At the one extreme every formula is an individual module and the problem solver never merges modules. In this case the result is equivalent to running BCP on formulae. As all the formulae may be pre-compilable, this may require no implicate construction at run time.

If the problem solver is exercising control to achieve completeness, we must examine more carefully when completeness is achieved for a particular $\mathcal{A}$. Just because some symbol is labeled **U** is no indication of incompleteness — no one guarantees that every literal or its negation should follow from $\mathcal{F} \cup \mathcal{A}$. However, if every clause is individually satisfied, then we know that the clause set is consistent and we can complete the labeling by arbitrarily changing every **U** to **T** or **F**. (Of course, this observation is implicit in the lazy algorithm which stops resolving clauses when they are satisfied.)

This last observation provides two fundamental techniques for coping with incompleteness. First, the problem solver can introduce additional assumptions to attempt to satisfy open constraints, in effect, performing a backtrack search. (This has the disadvantage of extending $\mathcal{A}$ which may not be desired.) Second, the problem solver controls which modules should be merged and in which order. Merging has two important effects: (a) merging can enable the construction of new implicates which yield relabelings, and (b) if each of the modules are either satisfied (we define a module to be satisfied if every one of its clauses is satisfied) or merged into one common unsatisfied module, then we know that BCP is complete. This tradeoff of whether to use backtracking or merging to construct a solution is analogous to the one faced by CSP [8; 18] solvers.

Both approaches to coping with incompleteness can be improved with various tactics. We focus here on tactics to improve the performance of merging. If an internal symbol is labeled **U**, then the modules which mention it are candidates for early merging. Whether or not this relabels the internal symbol, after the merge all clauses mentioning an internal symbol can be discarded. Modules sharing no symbols can be trivially merged as the implicates of the conjunction is the union of the antecedent implicates. If all modules are merged and BCP has not detected a violated clause, then by a slight extension of theorem 1 $\mathcal{F} \cup \mathcal{A}$ is satisfiable. When used in this way our TMS is yet another way to test for propositional satisfiability.

Our implementation also includes an automatic facility which systematically merges those two modules which would produce a module with the fewest number of symbols (determined directly by counting the non-internal symbols). This exploitation of locality often avoids intermediate implicate bloat.

8 Modeling

The user of this style of TMS must make a fundamental tradeoff whether all the formulae should be in one module (and hence be logically complete), or whether the formulae should all be in individual modules (more efficient but incomplete). Suppose all the formulae are in one module. For those symbols which were not provided any initial labels, the same set of implicates will now suffice for any labeling for them. This ideally matches the requirements of problem solving tasks which require the inputs to be changed while the input formulae remain constant. In other words, by computing the implicates we have made it easy to solve exponentially many problems via BCP on these implicates.

One clearcut example of this occurs in qualitative simulation. Typically qualitative analysis uses propagation to determine the qualitative behavior of a system, however, it is well known that simple propagation is incomplete and therefore that additional techniques are needed (feedback heuristics, feedback analysis, etc.) One such technique is the qualitative resolution rule [11] which assembles individual component models into larger assemblages so that (a) the entire device can be repeatedly simulated on different inputs by simple propagation alone and (b) larger devices can be analyzed by building it out of known assemblages.

Our TMS framework achieves the analogous effect. The qualitative resolution rule (sometimes called the qualitative Gauss rule) is implemented using our TMS. [11] presents an example where two pipes (Fig. 1) connected together produce a model for a single pipe. Consider the following instance of the qualitative resolution rule. Let x, y and z be qualitative quantities such that (we drop [...] when unambiguous),

$$x + y = 0, \quad -x + z = 0$$

From these two confluences we can infer the confluence,

$$y + z = 0. \tag{6}$$

(To those unfamiliar with qualitative physics this may not seem that surprising, but it is important to remember that qualitative arithmetic does not obey the usual field axioms and thus the equations cannot be manipulated as in conventional arithmetic.) The qualitative resolution rule is analogous to binary resolution. Two confluences can be usefully combined only if they share at most one symbol in common, otherwise the result is meaningless.

Our TMS achieves the effect of the qualitative resolution rule by conjoining the formulae of the two individual pipes. One way to expand $x + y = 0$ into clauses is to encode all the value combinations disallowed by the confluence:

$$\neg(x{=}+) \vee \neg(y{=}+),$$
$$\neg(x = +) \vee \neg(y = 0),$$
$$\neg(x = 0) \vee \neg(y = +),$$
$$\neg(x = 0) \vee \neg(y = -),$$
$$\neg(x = -) \vee \neg(y = 0),$$
$$\neg(x = -) \vee \neg(y = -).$$

Expanding $-x + z = 0$ into clauses includes:

$$\neg(x = -) \vee \neg(z = +),$$
$$\neg(x = -) \vee \neg(z = 0),$$
$$\neg(x = 0) \vee \neg(z = +),$$
$$\neg(x = 0) \vee \neg(z = -),$$
$$\neg(x = +) \vee \neg(z = 0),$$
$$\neg(x = +) \vee \neg(z = -).$$

If we add the clause,

$$(x = +) \vee (x = 0) \vee (x = -),$$

compute prime implicates and consider $\{x = +, x = 0, x = -\}$ internal symbols, then the result is exactly the prime implicates of the result of the qualitative resolution rule (i.e., of $y + z = 0$). This encoding might appear cumbersome, but the clauses are easily analyzed with BCP. As we have argued earlier, propagation on clauses (i.e., BCP) is efficiently implemented by following pointers and manipulating counters. Thus, by 'Assembling' the device, we obtain a set of prime implicates with which it is easy to determine a system's outputs from its inputs.

Dormoy [12] points out that applying the qualitative resolution rule sometimes produces a combinatorial explosion. This is analogous to the explosion that can occur in expanding a formula to its prime implicates. In his paper Dormoy proposes a joining rule for controlling this explosion. The joining rule applies the qualitative resolution only to components which share an internal variable — it is equivalent to our TMS heuristic of combining modules which share internal symbols.

Consider the two pipe problem of the introduction again. Suppose we know that $[dP_A] = [+]$ and $[dP_B] = [0]$. We have, in effect, two choices how to solve the problem. We could first inform the TMS of these values and then ask it to merge the modules of the two pipes; or we could first merge the two modules and then add these values. Although the answer $[dQ] = [+]$ remains the same, the resulting TMS data base is quite different. If we start with $[dP_A] = [+]$ and $[dP_C] = [0]$, then most of the prime implicate constructions can be avoided because these initial values provide initial BCP labels to 6 symbols (i.e., the symbols representing the possible qualitative values for dP_A and dP_B). On the other hand, if the modules are merged first without initial values, then all prime implicates are constructed, and although only a few of them are necessary to solve for the given inputs it is now much easier to solve problems when the inputs are changed.

Although computing all the prime implicates for a full device may be expensive, it often may be very useful to incur this cost. Once the prime implicates of a device are constructed, the input-output behavior is completely characterized. From the resulting data base of prime implicates one can construct the inputs from the outputs just as easily as outputs from the inputs without constructing any additional prime implicates. So the same data base can be efficiently utilized for a variety of distinct tasks.

9 Conclusion

We have shown a simple example of using our framework for simple qualitative physics analysis tasks. The applicability of these techniques extends to any problem-solving paradigm for which finite propositional encodings exist and there is some notion of locality in the original task which can be conveyed to the TMS. Many tasks for which TMS's are normally used are candidates for this approach. One obvious one which we are currently exploring is model-based diagnosis [5].

10 Acknowledgments

Daniel G. Bobrow, John Lamping, Olivier Raiman and Vijay Saraswat provided extensive insights on early drafts.

References

[1] Chang, C. and Lee R.C., Symbolic Logic and Mechanical Theorem Proving, (Academic Press, New York, 1973).

[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 Kaufman, 1987), 280–297.

[3] de Kleer, J. and Brown, J.S., A qualitative physics based on confluences, *Artificial Intelligence* **24** (1984) 7–83; also in: D.G. Bobrow (Ed.), *Reasoning About Physical Systems* (MIT Press and North Holland, 1985) 7-83; also in: J.R. Hobbs and R.C. Moore (Eds.), *Formal Models of the Common-Sense World* (Ablex, Norwood, NJ, 1985) 109–183.

[4] de Kleer, J., Extending the ATMS, *Artificial Intelligence* **28** (1986) 163–196.

[5] 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 Kaufman, 1987), 372–388.

[6] de Kleer, J., A practical clause management system, SSL Paper P88-00140, Xerox PARC.

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

[8] de Kleer, J., A comparison of ATMS and CSP techniques, *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, Detroit, MI (August 1989).

[9] de Kleer, J., A Clause Management System based on Boolean Constraint Propagation and clause synthesis, 1990.

[10] de Kleer, J., Compiling Devices, submitted for publication, 1990.

[11] Dormoy, J. and Raiman, O., Assembling a device, in: *Proceedings AAAI-88*, Saint Paul, Minn (1988) 330–336.

[12] Dormoy, J-L., Controlling qualitative resolution, in: *Proceedings AAAI-88*, Saint Paul, Minn (1988) 319–323.

[13] Forbus, K.D., Qualitative process theory, *Artificial Intelligence* **24** (1984) 85-168.

[14] Forbus, K.D., The qualitative process engine, University of Illinois Technical Report UIUCDCS-R-86-1288, 1986.

[15] Dowling and Gallier, Linear time algorithms for testing the satisfiability of propositional horn formulae, *Journal of Logic Programming* **3** 267–284 (1984).

[16] Kean, A. and Tsiknis, G., An incremental method for generating prime implicants/implicates, University of British Columbia Technical Report TR-88-16, 1988.

[17] Kohavi, Z., *Switching and Finite Automata Theory* (McGraw-Hill, 1978).

[18] Mackworth, A.K., Constraint satisfaction, *Encyclopedia of Artificial Intelligence*, edited by S.C. Shapiro, (John Wiley and Son, 1987) 205-211.

[19] McAllester, D., A three-valued truth maintenance system, S.B. Thesis, Department of Electrical Engineering, Cambridge: M.I.T., 1978.

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

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

[22] Reiter, R. and de Kleer, J., Foundations of Assumption-Based Truth Maintenance Systems: Preliminary Report, in: *Proceedings AAAI-87*, Seattle, WA (July, 1987), 183-188.

[23] Tison, P., Generalized consensus theory and application to the minimization of boolean functions, *IEEE transactions on electronic computers* **4** (August 1967) 446-456.