# Compiling Devices and Processes

**Johan de Kleer**
Xerox Palo Alto Research Center
3333 Coyote Hill Road, Palo Alto CA 94304 USA

**Unfinished Draft — Please do not redistribute**

February 26, 1990

**Abstract**

This paper presents a new approach for exploiting Truth Maintenance Systems(TMSs) in which the inference engine can convey locality in its knowledge to the TMS. This approach is ideally suited for systems which reason about the physical world because much of knowledge is inherently local, i.e., the constraints for a particular component or process usually only interact with constraints of physically adjacent components and processes. The new TMSs operate with a set of arbitrary propositional formulae 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 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 prime 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. We show that conveying locality to the TMS is an important strategy for qualitative physics problem solvers. For example, at a minimum formulae corresponding to a single component (or commonly occurring combinations) model should be compiled into prime implicates in order to minimize run-time cost. When confluence models are used, the results of using our TMS subsume those of the qualitative reasolution rule. 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 approach for exploiting Truth Maintenance Systems(TMSs) in which the inference engine can convey locality in its knowledge to the TMS. 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 local 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 atest that current TMSs have 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.

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 such 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) [2, 6]. Techniques like these are widely used in QPE [11, 12]. Unfortunately, these encodings tend to be extremely cumbersome. The TMSs which accept arbitrary clauses (such as Logic-Based Truth Maintenance Systems (LTMS) [2, 17, 18, 19]) seem to be more powerful because any propositional formula can easily be converted into a set of clauses by putting it into CNF[1].

Unfortunately, complete LTMSs based on clauses are never used because they are too inefficient. Instead, all common LTMS implementations use Boolean Constraint Propagation(BCP)[2, 17, 18, 19] 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 constraint. Consider the formula:

$$(x \Rightarrow (y \vee z)) \wedge (x \vee y \vee z) \tag{1}$$

If $y$ were false, then considering this formulae alone in isolation we can infer $z$ must be true (this can be seen by the fact that if $z$ were false, both conjuncts would evaluate to false). However, this information is lost in converting to the CNF form:

$$\neg x \vee y \vee z, \quad x \vee y \vee z. \tag{2}$$

Neither of these constraints can, individually, be used to infer $z$ from $\neg y$.

Consider QPE as an example [11, 12]. QPE encodes every qualitative process model as a set of formulae which are eventually encoded as a set of ATMS horn clauses. Within the inference engine, this set of horn clauses represents a fixed local module, but within the

ATMS these modules are independent. As some of these clauses are non-Horn the basic ATMS algorithms are incomplete, and the ATMS is therefore incapable of making 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 any substantial performance degradation. 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 operate with a set of arbitrary propositional formulae and use general BCP to answer queries whether a particular literal follows from the constraints which although usually applied to clauses can be applied to arbitrary formulae as well. As input our 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 which modules should be conjoined and thus which prime implicates should be constructed such that even when logical completeness is required that relatively of the prime implicates of the entire formula set need be constructed. This approach has been implemented and tested both with ATMSs and LTMSs.

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:

$$[dP_A] - [dP_B] = [dQ_{AB}],$$

$$[dP_B] - [dP_C] = [dQ_{BC}],$$

$$[dQ_{AB}] = [dQ_{BC}].$$

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. If the individual qualitative equations are converted to their propositional equivalents for a TMS (as many qualitative physics systems do), the flows 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 devices, this combining can be done once in the model library. To compile this combination, our TMS merges the propositional

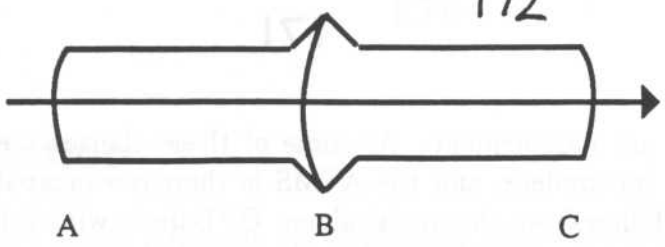


Figure 1: Assembling the qualitative models of the two joined pipes is equivalent to merging the two formulae modeling the two pipes.

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 $[dQ_{AB}] = [dQ_{BC}] = [+]$ far more efficiently than before (i.e., immediately). A device can always be analyzed by first compiling it without knowledge of any input or outputs. However, compiling a full device 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 constraints it is supplied — it only compiles enough to answer the query from 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 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 7 expands on these observations and analyzes its relationship with the qualitative resolution rule [9, 10].

## 2 BCP on formulae and clauses

As our approach draws deeply on the intuitions underlying BCP, we synopsize it here. (Note that BCP achieves similar results to unit resolution.) BCP operates on a set of propositional formulae (not just clauses) $\mathcal{B}$ in terms of propositional symbols $\mathcal{S}$. A formula is defined in the usual way with the connectives $\neg, \Rightarrow, \equiv, \vee, \wedge$ and $tax$. ($tax$ is an extremely useful connective requiring that exactly one of its arguments be true.)

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**. All remaining symbols are initially labeled **U**. The reason for distinguishing $\mathcal{A}$ from $\mathcal{B}$ is that $\mathcal{B}$ 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 the formulae $\mathcal{B} \cup \mathcal{A}$. 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):

- The labeling *satisfies* the constraint: for every completion of the current labeling the constraint is true. For example, the labeling $x$ to be **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** 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 **F**.
- 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 the constraints are clauses, then BCP can be implemented efficiently. In particular, we store a count with each clause indicating the number of symbols mentioned by the clause whose current label is opposite to how it appears in 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 [6] for details.) BCP on clauses is equivalent to the circuit value problem and therefore is P-complete (see also [13]). 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:

$$\neg x \vee y \vee z, \quad x \vee y \vee z. \tag{3}$$

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 constraints

The previous example (the encoding of equation (1) shows that running BCP on the original formulae is not the same as running BCP on the clauses produced by converting the formulae to CNF. (BCP on the original formulae is always stronger.) Hence, we cannot directly use the efficient BCP algorithms that have been developed for clauses for arbitrary constraints and no correspondingly efficient BCP algorithm is known for arbitrary formulae. This section shows that if each individual constraint is encoded by its prime implicates[14, 15, 20], then BCP on the resulting clauses is equivalent to running BCP on the original formulae.

Clause $A$ is subsumed by clause $B$ if all the literals of $B$ appear in $A$. An implicate of a set of formulae $\mathcal{B}$ is a clause which logically follows from $\mathcal{B}$. A prime implicate of a set of formulae $\mathcal{B}$ is an implicate of $\mathcal{B}$ which is not subsumed by some other implicate of $\mathcal{B}$. Using these definitions, the following two theorems are key to an efficient implementation of BCP on constraints:

**Theorem 1** *Suppose that the set of clauses $\mathcal{I}$ are the set of prime implicates of some set of propositional formulae, then BCP on $\mathcal{I}$ is logically complete.*

*Proof.* We presume BCP is sound. We must show that the theorem holds for any initial set of assumptions $\mathcal{A}$. Suppose that literal $x$ logically follows from $\mathcal{I}$ and assumptions $A_1, \ldots, A_n$ (possibly none). If $x$ follows from $A_1, \ldots, A_n$ then the formula,

$$A_1 \wedge \cdots \wedge A_n \Rightarrow x,$$

which in clausal form is,

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

must be an implicate of $\mathcal{I}$ and thus subsumed by some clause of $\mathcal{I}$.

**Theorem 2** *Given a set of propositional formulae $\mathcal{B}$ and $\mathcal{I}$ is the union of the prime implicates of each of the formulae of $\mathcal{B}$, then BCP on $\mathcal{B}$ produces the same labeling as BCP on $\mathcal{I}$.*

*Proof.* We presume BCP is sound. As each formula is individually replaced by its prime implicates, BCP on $\mathcal{I}$ cannot label any symbol that BCP on $\mathcal{B}$ does not. We do the reverse direction by proof by contradiction. Suppose BCP on formula $b \in \mathcal{B}$ labels symbol $s$ while BCP on the prime implicates of $b$ does not. If $s$ is labeled $\mathbf{T}$ let $x$ be the literal $s$ and if $s$ is labeled $\mathbf{F}$ let $x$ be the literal $\neg s$. Given BCP on $b$ labels $s$ it must be the case that,

$$A_1 \wedge \cdots \wedge A_n \wedge x_1 \wedge \cdots \wedge x_m \Rightarrow x,$$

follows from $b$ alone where the $x_i$ are literals involving symbols of $b$ and $A_i$ are assumptions. Therefore, the clause,

$$\neg A_1 \vee \cdots \vee \neg A_n \vee \neg x_1 \vee \cdots \vee \neg x_m \vee x,$$

is an implicate of $b$ alone. Therefore, it either is or is subsumed by a prime implicate of $b$. As this prime implicate necessarily contains $x$, BCP on this implicate must have labeled $x$ true.

The first theorem tells us that we can make BCP complete if we need to. The second theorem tells 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.

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

$$(x \Rightarrow (y \vee z)) \wedge (x \vee y \vee z), \tag{4}$$

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 is illustrates that there may be fewer prime implicates than the conjuncts in the conventional CNF. 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:

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

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.

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. In general, converting individual formulae to their individual prime implicate form is far more effective.

There are a variety of different algorithms for computing prime implicates (see [5, 6, 8, 14, 20, 21]). Stripped from 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 complementary literals and construct a resulting clause with both those literals removed. All subsumed clauses are removed. This process continues until no new unsubsumed clause is producible.

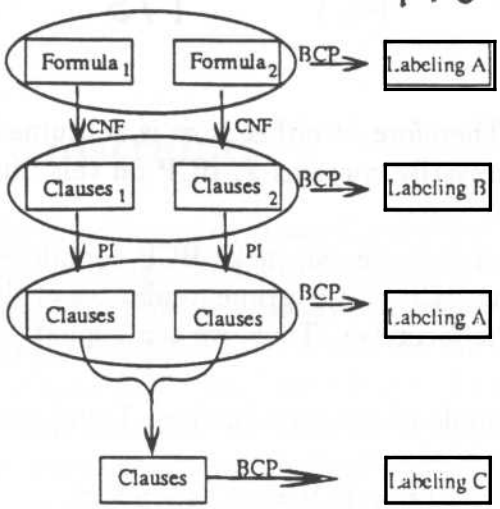


Figure 2: This figure illustrates the different ways BCP can be used. BCP on arbitrary formulae (expensive) produces labeling A. If the formulae encoded as their CNF clauses, then an efficient clausal BCP produces an (unfortunately weaker) labeling B. If formulae are individually converted into their 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.

# 4 Limiting prime implicate construction

There are a number of important observations which reduce the complexity of dealing with prime implicates.

Prime implicates need to be computed only when a formula is added. No recomputation is required if more assumptions are added or retracted. Therefore, the major cost is paid up front when the a formula is added to $\mathcal{B}$ and not when $\mathcal{A}$ is changed.

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 [11, 12] most processes are instantiated with the same fixed set of formulae (but with different symbols). Hence, the schemas for prime implicates for each model in the library can be constructed a priori, and prime implicate constructions can be avoided run time.

Sometimes it becomes necessary to conjoin two sets of formulae at run time. We can exploit the observation that the prime implicates of a conjunction of two formulae is the same as the prime implicates of the union of the prime implicates of the two individual formulae. Starting with the two initial sets of formulae is usually far more efficient than starting from scratch.

We exploit BCP labels to construct prime implicates lazily. But we must distinguish the different reasons a symbol can be labeled **T**/**F**: (1) a symbol can be labeled because it follows from $\mathcal{B}$ alone (we call this a *fixed* label), and (2) a symbol can be labeled because it follows from $\mathcal{B} \cup \mathcal{A}$ (we call this a *variable* label). The important difference is that the fixed labels cannot change if assumptions are added or retracted. If a fixed literal label satisfies a clause, then that clause is removed from $\mathcal{B}$. 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. Therefore our implementation only resolves

two clauses if the complementary literals are labeled **U**. For efficiency, as soon as any new clause is constructed, BCP is run in order to try to avoid consensus constructions. Note that as variable labels change clauses which were previously set aside need to be reanalyzed to compute further prime implicates.

A single propositional formula can 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 prime implicates mentioning that symbol can be discarded without affecting the functionality of the TMS. As a result BCP need not stumble over these needless clauses.

# 5 Specifications

In this section we briefly summarize the basic TMS transactions within our framework.

`(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. Section 2 outlines the allowed connectives. `internal-symbols` is a set of internal symbols which are guaranteed not to appear again. Therefore, after computing prime implicates, all references to these symbols can be discarded.

`(load-formula schema)`: Used at run time. This takes the prime implicate schemas and communicates them to the TMS. We call the set of prime implicates for a formula a *module*. It returns a description of the module.

`(add-formula formula)`: Used at run time. This adds an individual formula to the TMS. This is equivalent to using `compile-formula` and `load-formula` in succession.

`(merge-modules module1 module2)`: Used at run time. This tells the TMS to conjoin the two modules, by computing the prime implicates of the combination.

`(enable-assumption symbol label)`: Used at run time. This labels the symbol **T** or **F**.

`(retract-assumption symbol)`: Used at run time. This removes the initial label for the symbol. Note that the symbol might retain a non-**U** label if it can be derived via BCP from other clauses.

`(label? symbol)`: Used at run time. This returns the label for the symbol.

`(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.

# 6 Coping with incompleteness

BCP is, in general, incomplete. But we have to be careful to determine when this incompleteness poses a difficulty. Just because some symbol is labeled **U** is no indication of

incompleteness — no one guarantees that enough formulae are provided to force every symbol to be **T** or **U**. However, if every formula is individually satisfied by some labeling, then we know that the clause set is consistent and we can complete the labeling by arbitrarily changing every **U** to **T** or **F**. These observations provide two fundamental techniques for coping with incompleteness. (We define a module to be satisfied if every one of its clauses is satisfied.) First, the problem solver can introduce additional assumptions to attempt to satisfy open constraints, in effect, performing a backtrack search. Second the problem solver can merge modules. Merging has two important effects: (a) merging can enable the construction of new prime implicates which yield relabelings, and (b) if each of its modules are either 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 [7, 16] 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 the internal symbol mentioning it can be discarded. Modules sharing no symbols can be trivially merged as the prime implicates of the conjunction is the union of the antecedent prime implicates. However, that if there are variable labels which are subsequently changed after a merge, the fact that the merge occurred causes additional work.

# 7 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 their individual modules. For pragmatic reasons, it is usually far better to have individual modules, however, there are applications where forming larger modules is extremely useful. For those symbols which were not provided any initial labels, the same set of prime implicates will now suffice for any initial labeling. This ideally matches the requirements of problem solving tasks which require the inputs to be changed and with the input formulae remaining constant. In other words, by computing the prime implicates we have made it easy to solve $2^n$ different problems via BCP.

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 [9] which assembles individual component models into larger assemblages so that (a) the entire the device is 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. In fact, the qualitative resolution rule (sometimes called the qualitative Gauss rule) can be implemented using our TMS. [9] 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,

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

From these two confluences we can infer the confluence,

$$y + z = 0. \tag{5}$$

(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 the 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 acchieves the effect of the qualitative resolution rule by conjoining the formulae of the two individual pipes. Expanded into prime clauses $x+y=0$ includes: $\neg(x{=}+) \vee \neg(y{=}+), \neg(x=+) \vee \neg(y=0), \neg(x=0) \vee \neg(y=+), \neg(x=0) \vee \neg(y=-), \neg(x=-) \vee (y=0), \neg(x=-) \vee \neg(y=-)$. Expanding $-x+z=0$ into clauses includes: $\neg(x=-) \vee \neg(z=+), \neg(x=-) \vee \neg(z=0), \neg(x=0) \vee \neg(z=+), \neg(x=0) \vee \neg(z=-), \neg(x=+) \vee (z=0), \neg(x=+) \vee \neg(z=-)$. 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 result is nevertheless efficient. 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 [10] 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 attempting to combine modules which share internal symbols.

The TMS formulation provides also admits another approach. 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 two 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 they provide initial BCP labels to 6 assumption symbols. On the other hand, if the modules are merged first, 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 where 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

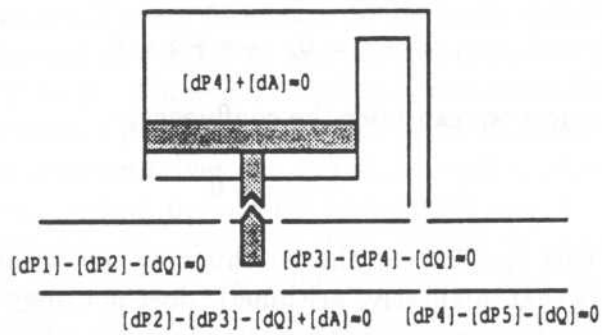


Figure 3: Constructing a composite model of the pressure regulator

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.

Consider the pressure regulator (Fig. 3) as an example. One version of the pressure regulator is described by the confluences in the figure (from [10]). Converting into propositional formulae and providing no initial labels, our TMS yields 2814 prime implicates (many of which are trivial). At first sight this seems like an awful lot of information, but these 2814 prime implicates contain a very large amount of information for performing a variety of tasks on the device and thus it makes sense to construct these implicates if we are repeatedly performing tasks on the same device. For example, many of the implicates are of the form:

$$([dP_1] \neq +) \wedge ([dP_5] \neq -) \Rightarrow ([dA] \neq +).$$

Often we are only interested in those values which follow from some other, and not such inequalities. Of the 2814 prime implicates, 496 are such definite clauses. If we regard all except $P_1, P_5$ and $Q$ as internal variables, there remain 21 prime implicates. Finally, if we know that $[dP_1] = +$ and $[dP_5] = 0$, then one clause remains. Equivalently, of the 2814 prime implicates, only 50 do not contain internal variables, and 21 of these are definite clauses. Thus we see that the input-output behavior of the pressure regulator can be compiled into relatively few very simple clauses. As our TMS compute prime implicates lazily there is never any reason to compile the whole device except, perhaps, for the model library. Instead, as each of the possible givens (usually inputs) are supplied, the TMS computes enough prime implicates to answer the query. Only after all input-output combinations (including negations and disjunctions) have been explored will the TMS have constructed all the prime implicates. (This technique may not be as successful for all devices because in the worst case, given $n$ input-output variables, we might need $2^n$ clauses to characterize its behavior.)

In the case of [9, 10] it makes not practical difference whether the qualitative resolution rule is used or our TMS — the effect is the same. However, the prime implicate formulation is more general, these same advantages accrue to any formulation of qualitative arithmetic and, indeed, to any problem solver which uses a TMS.

# 8 Acknowledgments

Daniel G. Bobrow, John Lamping and Olivier Raiman 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., A practical clause management system, SSL Paper P88-00140, Xerox PARC.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[20] 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.

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

[22] Williams, B.C., MINIMA A symbolic approach to qualitative algebraic reasoning, in: *Proceedings AAAI-88*, Saint Paul, Minn (1988) 264–270.