DESIGN AND IMPLEMENTATION OF MULTIPLE-CONTEXT TRUTH MAINTENANCE SYSTEM WITH BINARY DECISION DIAGRAM

Hiroshi G. Okuno, **Osamu Shimokuni,** and **Hidehiko Tanaka**
NTT Basic Research Laboratories
Nippon Telegraph and Telephone Corporation
3–1 Morinosato-Wakamiya, Atsugi
Kanagawa 243-01, JAPAN
Graduate School of Engineering
The University of Tokyo
7–3–1 Hongo, Bunkyo-ku
Tokyo 113, JAPAN
okuno@nuesun.brl.ntt.jp, osamus@ipl.t.u-tokyo.ac.jp, and tanaka@mtl.t.u-tokyo.ac.jp

ABSTRACT

Implicit enumeration of prime implicates in Truth Maintenance System (TMS) is investigated. CMS (Clause Management System), an extension of Assumption-based TMS (ATMS), that accepts any type of justification has a burden to compute all prime implicates, since its complexity is NP-complete. To improve the performance of multiple-context TMS such compact representation of boolean functions. In this paper, we propose a BDD-based Multiple-context TMS (BMTMS) and present the design and implementation of interface between TMS and BDD. The interface provides high level specifications of logical formulas, and has mechanisms to schedule BDD commands to avoid combinatorial explosions in constructing BDDs. In BMTMS, most TMS operations are carried out without enumerating all prime implicates.

1 INTRODUCTION

The capability of thinking with explicit multiple alternatives is required by sophisticated problem solving systems such as qualitative simulation, multi-fault diagnosis or non-monotonic reasoning. Since a consistent database (data collection) is referred to as *context*, the above requirement can be paraphrased as *multiple context reasoning*. Multiple-context reasoning is, in general, superior to single-context reasoning in switching contexts or comparing the results between context [deK86a].

A problem solving system consists of an inference engine and a truth maintenance system (TMS). The inference engine introduces hypotheses and makes inferences, while TMS records hypotheses as *assumptions* and inferences as *justifications*. TMS maintains the inference process and enables the inference engine to avoid futile or redundant computations [FdK93]. TMSs are classified according to two properties. One property is whether they provide a single or a multiple context and the other is whether they accept Horn clauses or general clauses as justifications. This classification is summarized in Table 1.

Justification-based TMS (JTMS) is a single-context TMS that accepts only a Horn clauses as a justification. JTMS is very popular so far, because JTMS runs very efficiently with backtracking mechanism. The satisfiability of a set of Horn clauses, or the assignment of variables that satisfies the set of clauses, can be solved in a linear time of the number of variables [Dow84]. However, JTMS cannot perform efficiently reasoning with alternatives.

Logic-based TMS (LTMS) is a single-context TMS that accepts general clauses including disjunction or negation. An efficient algorithm for LTMS is a *Boolean constraint Propagation (BCP)*, a kind of intelligent backtracking mechanism. First, the conjunction of justifications is converted to a conjunctive normal form (CNF, or Product of Sum, POS) and decomposed to a set of clausal forms. BCP takes a set of clausal forms and labels each node consistently by backtracking. BCP with a set of clausal forms is complete for Horn clauses, but not for general clauses. This incompleteness is caused by being information dropped in conversion of a clause to a set of clausal forms. To

Table 1 Classification of TMS

context / JUSTIFICATION	*single context*	*multiple context*
HORN CLAUSES	Justification-based TMS (JTMS)	Assumption-based TMS (ATMS)
GENERAL CLAUSES	Logic-based TMS (LTMS)	Clause Management System (CMS)

recover completeness, all prime implicates should be added to the set of clausal forms, but the whole performance of such a BCP deteriorates because the total number of prime implicates is generally very large. For example, the modeling of two containers connected by a valve with seven observation points produces 2,814 prime implicates [FdK93]. In addition, the computational complexity of enumerating prime implicates is NP-complete.

A multiple-context TMS is, in general, superior to a single-context TMS in its capabilities and efficiency in seeking all solutions [deK86a]. In particular, *Assumption-based TMS (ATMS)*, a multiple-context TMS that accepts only Horn clauses as justifications can be implemented quite efficiently by compiling justifications into a network [deK86a; Oku90]. A datum of inference engine is represented by a node and its belief status is maintained by a label. The labels are computed incrementally by using the network. However, ATMS has a limited power of expression. When the problem solving system wants to handle a general clause, it must be encoded to a set of Horn clauses with special constructs. And some additional routines are needed to validate the completeness of TMS operations [deK86b]. This type of encoding sometimes gives rise to the degradation of the total performance of the system.

Clause management system (CMS) is a multiple-context TMS that accepts general clauses including disjunction and negation [RdK87]. In CMS, the label of a node is computed via minimal support [RdK87]. Let Σ be the set of all the justifications and PI be a prime implicates of Σ. The minimal support for the clause C is computed as follows:

$$MinSup(C, \Sigma) = \{S | S \in \Delta(C, \Sigma), \text{ S is minimal}\} \quad (1)$$
$$\Delta(C, \Sigma) = \{PI - C | PI \cup C \neq \{\}\}$$

Note that the computation of the minimal support requires the enumeration of all prime implicates. This is the main factor of intrinsic poor performance of CMS.

We have been studying an implicit enumeration of prime implicates. Recently, a Binary Decision Diagram (BDD) is proposed, which is a compact representation and provides efficient manipulations of boolean functions [Bry92]. Recent techniques with BDDs can generate more than 10^{10} prime implicates [LCM92]. Since BDDs can represent all solutions simultaneously, it is reasonable to use BDDs to implement a multiple-context TMS. In this paper, we propose a *BDD-based Multiple-context TMS (BMTMS)* to exploit two ways:

1) Implementing efficient methods for enumerating all prime implicates, and
2) Implementing TMS operations without enumerating prime implicates explicitly.

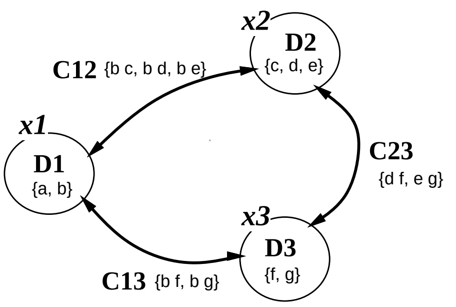


FIGURE 1 A simple constraint satisfaction problem

Madre *et al.* implemented ATMS by a variant of BDD called TDG (Typed Decision Graphs), but only the first issue was addressed [MC91].

The rest of the paper is organized as follows: In Section 2, the issues of CMS are identified. In Section 3, BDDs are explained and the issues in applying BDDs to multiple-context TMS is investigated. In Section 4, the BMTMS is proposed and its details are described. In Section 5, some capabilities of the BMTMS are demonstrated. The evaluation results of the BMTMS, and conclusions are given in Section 6 and Section 7, respectively.

2 IDENTIFYING THE ISSUES OF CMS

In this section, the drawbacks of the ATMS and the problems of CMS are demonstrated by using a a simple problem [deK89] (hereafter, *the simple problem*) shown in Figure 1. It has three variables, x_1, x_2, and x_3. The domain of each variable is $D_1 = \{a, b\}$, $D_2 = \{c, d, e\}$, and $D_3 = \{f, g\}$, respectively. For a pair of variables, x_i and x_j, a constraint on them, $C_{i,j}$, is given as the set of permissible combinations of values. They are $C_{12} = \{b\,c, b\,d, b\,e\}$, $C_{13} = \{b\,f, b\,g\}$, and $C_{23} = \{d\,f, e\,g\}$.

2.1 Encoding by ATMS

In ATMS, an inference-engine's datum is represented by a *TMS-node*, and an assumption is a special kind of TMS-node. A justification has a set of *antecedents* and a single *consequence*, all of which are TMS-nodes. In other words, a justification is a propositional Horn clause. An *environment* is represented by a set of assumptions. A contradictory environment is called *nogood*. Each TMS-node has a *label*, or a set of *environments*, under which the node is proved valid.

The simple problem cannot encoded by ATMS itself, because it contains disjunctions and thus is not a Horn clause. The encoding techniques proposed by de Kleer [deK89] is used for the encoding. The inference engine gives the following data to ATMS:

- A propositional symbol, $x_{i:v}$. A symbol $x_{i:v}$ means that a variable x_i has a value v. Thus, $x_{1:a},\ x_{1:b},\ x_{2:c},\ x_{2:d},\ x_{2:e},\ x_{3:f},\ x_{3:g}$.
- A set of justifications, each of which encodes the domain of a variable.
  $x_{1:a} \vee x_{1:b},\ x_{2:c} \vee x_{2:d} \vee x_{2:e},\ x_{3:f} \vee x_{3:g}$.
  Since these clauses are not Horn ones, a disjunction is encoded by using the `choose` predicate as follows:

  `choose`$\{x_{1:a}, x_{1:b}\}$, `choose`$\{x_{2:c}, x_{2:d}, x_{2:e}\}$, `choose`$\{x_{3:f}, x_{3:g}\}$.
- A set of justifications, each of which states that each variable have only one value.
  $\neg x_{1:a} \vee \neg x_{1:b},\ \neg x_{2:c} \vee \neg x_{2:d},\ \neg x_{2:c} \vee \neg x_{2:e},\ \neg x_{2:d} \vee \neg x_{2:e},\ \neg x_{3:f} \vee \neg x_{3:g}$.
  These are encoded by using the `nogood` predicate as follows:

  `nogood`$\{x_{1:a}, x_{1:b}\}$, `nogood`$\{x_{2:c}, x_{2:d}\}$, `nogood`$\{x_{2:c}, x_{2:e}\}$, `nogood`$\{x_{2:d}, x_{2:e}\}$, `nogood`$\{x_{3:e}, x_{3:f}\}$.
- A set of justifications, each of which encodes an inhibited pair specified by a constraint.
  $\neg x_{1:a} \vee \neg x_{2:c},\ \neg x_{1:a} \vee \neg x_{2:d},\ \neg x_{1:a} \vee \neg x_{3:e},\ \neg x_{2:d} \vee \neg x_{3:g},\ \cdots$.
  These are encoded as follows:

  `nogood`$\{x_{1:a}, x_{2:c}\}$, `nogood`$\{x_{1:a}, x_{2:d}\}$, `nogood`$\{x_{1:c}, x_{3:e}\}$, `nogood`$\{x_{2:d}, x_{3:g}\}$, $\cdots$.

To find solutions that satisfy these justifications, ATMS uses a label update algorithm. Since ATMS is complete for Horn clauses but not for arbitrary logical formula, ATMS fails to calculate the correct labels for the above encodings. To attain completeness, meta rules concerning `choose` and `nogood` are needed. These meta rules perform hyperresolutions for these predicates. Some examples of hyperresolutions are listed below:

$$\frac{\begin{array}{l}\texttt{choose}\{x_{3:f}, x_{3:g}\}\\ \texttt{nogood}\{x_{2:c}, x_{3:f}\}\\ \texttt{nogood}\{x_{2:c}, x_{3:g}\}\end{array}}{\texttt{nogood}\{x_{2:c}\}}$$

$$\frac{\begin{array}{l}\texttt{choose}\{x_{2:c}, x_{2:d}, x_{2:e}\}\\ \texttt{nogood}\{x_{2:c}\}\end{array}}{\texttt{choose}\{x_{2:d}, x_{2:e}\}}$$

The final solutions are

$$(x_{1:b} \wedge x_{2:d} \wedge x_{3:f}) \vee (x_{1:b} \wedge x_{2:e} \wedge x_{3:g}).$$

This encoding works well for the simple problem, but does not work for all CSPs. de Kleer pointed out that the meta rules correspond to local consistency algorithms [deK89]. Applying a meta rule that corresponds to arc-consistency algorithm is very expensive, and thus the completeness may be fulfilled not at all time.

2.2 Encoding by CMS

The simple problem is directly encoded by logical expressions. In CMS encoding, the same propositional symbols as ATMS are used. The domain of each variable is encoded as $X1$, $X2$, and $X3$. The select-one type constraint that each variable has only one value is encoded by $C1$, $C2$, and $C3$. The constraints between two variables are directly encoded.

$$\begin{aligned}
X1 &= x_{1:a} \vee x_{1:b}\\
X2 &= x_{2:c} \vee x_{2:d} \vee x_{2:e}\\
X3 &= x_{3:f} \vee x_{3:g}\\
C1 &= (x_{1:a} \wedge \neg x_{1:b}) \vee (\neg x_{1:a} \wedge x_{1:b})\\
C2 &= (x_{2:c} \wedge \neg x_{2:d} \wedge \neg x_{2:e}) \vee (\neg x_{2:c} \wedge x_{2:d} \wedge \neg x_{2:e})\\
&\quad \vee (\neg x_{2:c} \wedge \neg x_{2:d} \wedge x_{2:e})\\
C3 &= (x_{3:f} \wedge \neg x_{3:g}) \vee (\neg x_{3:f} \wedge x_{3:g})\\
C12 &= (x_{1:b} \wedge x_{2:c}) \vee (x_{1:b} \wedge x_{2:d}) \vee (x_{1:b} \wedge x_{2:e})\\
C13 &= (x_{1:b} \wedge x_{3:f}) \vee (x_{1:b} \wedge x_{3:g})\\
C23 &= (x_{2:d} \wedge x_{3:f}) \vee (x_{2:e} \wedge x_{3:g})\\
F &= X1 \wedge X2 \wedge X3 \wedge C1 \wedge C2 \wedge C3\\
&\quad \wedge C12 \wedge C13 \wedge C23 \qquad (2)
\end{aligned}$$

The simple problem is encoded as F. The prime implicates of F are

$\neg x_{2:d} \vee \neg x_{3:g},\ x_{2:d} \vee x_{3:g},\ \neg x_{3:f} \vee x_{2:d},\ x_{3:f} \vee \neg x_{2:d},\ \neg x_{2:e} \vee \neg x_{2:d},\ x_{3:f} \vee x_{2:d},\ \neg x_{2:e} \vee \neg x_{2:d},\ \ldots$

Let *Goal* be the goal literal. The simple problem is thus encoded as $F \supset \mathit{Goal}$. The label of the *Goal* literal is computed by Equation (1) and the following results are obtained:

$$(x_{1:b} \wedge x_{2:e} \wedge x_{3:g} \wedge \neg x_{1:a} \wedge \neg x_{2:c} \wedge \neg x_{3:d} \wedge \neg x_{3:f}) \supset \mathit{Goal}$$
$$(x_{1:b} \wedge x_{2:d} \wedge x_{3:f} \wedge \neg x_{1:a} \wedge \neg x_{2:e} \wedge \neg x_{2:c} \wedge \neg x_{3:g}) \supset \mathit{Goal}$$

Thus, the solution is obtained as the label of the literal *Goal*, that is,

$\{x_{1:b} \wedge x_{2:e} \wedge x_{3:g} \wedge \neg x_{1:a} \wedge \neg x_{2:c} \wedge \neg x_{3:d} \wedge \neg x_{3:f},\ x_{1:b} \wedge x_{2:d} \wedge x_{3:f} \wedge \neg x_{1:a} \wedge \neg x_{2:e} \wedge \neg x_{2:c} \wedge \neg x_{3:g}\}$.

Since the labels in ATMS satisfies four properties — soundness, completeness, consistency, and minimality —, they can be computed incrementally and efficiently

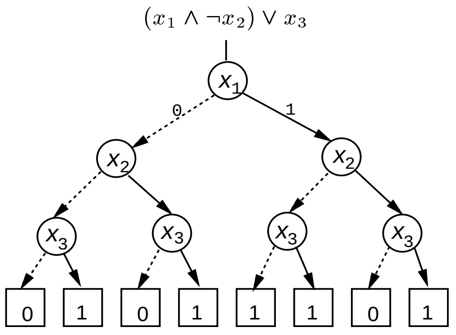
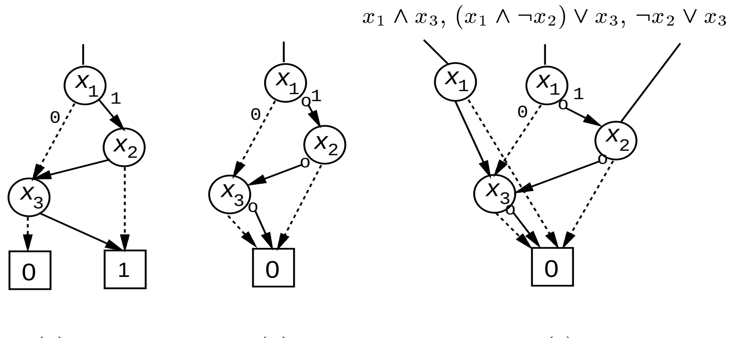


FIGURE 2 Tree representation


$x_1 \wedge x_3$, $(x_1 \wedge \neg x_2) \vee x_3$, $\neg x_2 \vee x_3$
x_1
x_2
x_3
0
1

(a) ROBDD (b) Negated edges (c) SBDD

FIGURE 3 ROBDD and its variants

[deK86a]. On the other hand, the labels are computed in batch in CMS by enumerating all prime implicates, because no algorithms are proposed so far to get all prime implicates incrementally. The label of a node plays a key role for efficient processing in ATMS, while it is one source of inefficiency in CMS. To cope with CMS's inefficiency due to enumeration of prime implicates, we exploit BDDs in the following two ways in the next section.

1) TMS operations without using labels, and
2) efficient enumeration of prime implicates.

3 Binary Decision Diagram (BDD)

3.1 Representing logical functions

Akers proposed a Binary Decision Diagram (BDD) [Ake78] as a compact representation of boolean (logical) functions, and Bryant invented efficient algorithms for manipulating them [Bry86]. Boolean functions can be represented by a tree. For example, the logical function, $((x_1 \wedge \neg x_2) \vee x_3)$, is represented by a tree, where a node represents a variable and two kinds of leaves represent 0 and 1 (Figure 2). Two edges of each node are called *0-edge* and *1-edge*, which represent the value when the variable associated with the node takes 0 and 1, respectively.

By fixing the order of variables, say, x_1, x_2, x_3 (where x_1 is the uppermost variable), sharing leaves and duplicate nodes, and removing redundant nodes, this tree can be transformed into an ROBDD (Reduced Ordered BDD) (Figure 3-(a)). An ROBDD gives a canonical form of a boolean function, and thus equivalent functions are represented by the same ROBDD. Various techniques have been invented to reduce the size of BDDs. One is a negated edge, which is indicated as a small circle attached to an edge (Figure 3-(b)). Another example is a Shared BDD (SBDD), which shares BDDs among many functions. Three functions share BDDs (Figure 3-(c)). Hereafter, BDD refers to SBDD with negated edges.

A BDD is constructed incrementally by the *apply* function instead of reducing a tree representation. Suppose *var-order* returns the order of a variable. Then, the apply function is defined recursively as follows:

apply(bdd1,bdd2,operation) =

- if *var-order*(bdd1's root) = *var-order*(bdd2's root), create a node by performing the operation to get a 0-edge and 1-edge.
- if *var-order*(bdd1's root) > *var-order*(bdd2's root), create a node with *apply*(bdd1's 0-edge, bdd2, operation) for the 0-edge, and *apply*(bdd1's 1-edge, bdd2, operation) for the 1-edge.
- otherwise, *apply*(bdd2, bdd1, operation).

This definition can be explained as performing logical operations on the Shannon expansion. Suppose that the Shannon expansions of functions f and g are as follows:

$$f = (\neg x \wedge f_{|x=0}) \vee (x \wedge f_{|x=1}), \text{ and}$$
$$g = (\neg x \wedge g_{|x=0}) \vee (x \wedge g_{|x=1}).$$

The logical AND of f and g, $f \wedge g$, is computed as

$$(\neg x \wedge (f_{|x=0} \wedge g_{|x=0})) \vee (x \wedge (f_{|x=1} \wedge g_{|x=1})).$$

The computational cost of the logical AND is in the order of the product of the numbers of nodes of both BDDs. Most functions can be executed in the same order. In addition, the *apply* function can be implemented efficiently by using a node hash table and a result cache (hash table). Since the result cache stores recent results of *apply* functions, redundant computations may be avoided if the cache hits. Since BDD is a compact representation of boolean functions and many usual boolean operations are performed efficiently with BDDs, BDD is commonly used in VLSI CAD systems.

Table 2 Comparison of size of BDDs

Constraint ordering	(2)	(3) *
Variable ordering	Best	Best *
Maximum size of intermediate BDDs	101	31
Size of final BDD	24	30

* Both constraint and variable ordering are determined by the CCVO heuristics.

The simple problem of the previous section can be solved directly by creating the BDD for F by using Equation (2). However, this BDD is an implicit representation of solutions and all the paths from the root of the BDD to the leaf of 1 should be enumerated to get the explicit representations of solutions.

3.2 Issues in using BDD

Applying BDDs to TMS has four main issues:

1) **Variable ordering**
2) **Constraint ordering**
3) **Encoding by logical functions**
4) **Compatibility with existing systems**

The number of nodes of a BDD, or *the size* of BDD, is determined by variable order. The minimum (best) size of 2-level AND-OR circuit of n variables is in the linear order of n, while the worst size is in the order of 2^n. Therefore the optimal variable order under which the size of BDD is minimum is very important. However, the computational complexity of finding the optimal variable order is NP-complete. This is well-known problem and many heuristics to find a near-optimal variable ordering is proposed.

However, constraint order is essential to avoid combinatorial explosions [Oku94]. In creating the BDD for F, nine constraints, $X1$, ..., $C23$, are used. Consider two constraint orderings below:

$$X1 \wedge X2 \wedge X3 \wedge C1 \wedge C2 \wedge C3 \wedge C12 \wedge C13 \wedge C23 \qquad (2)$$

$$X1 \wedge C1 \wedge X2 \wedge C2 \wedge C12 \wedge X3 \wedge C3 \wedge C13 \wedge C23 \qquad (3)$$

The size of BDDs created by (2) and (3) is shown in Table 2. The size of intermediate BDDs is affected by constraint order. The importance of constraint ordering is discovered by applying BDDs to solve combinatorial problems. Okuno proposed the CCVO (Correlation-based Constraint and Variable Ordering) heuristics to find a near-optimal constraint ordering. The CCVO first calculates the correlation of variables between constraints and then determines constraint ordering. Variable ordering is determined the first time a variable is used by constraint ordering.

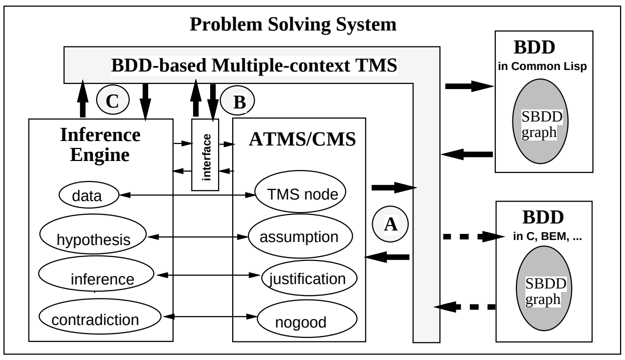


FIGURE 4 Structure of BMTMS

Another problem of constraint ordering occurs in applying BDDs to TMS, because constraints or justifications are given incrementally. Therefore, a new method of determining constraint ordering is required.

The third issue is concerning the expressive power of logical functions. In this paper, arithmetic boolean functions are used [Min93a]. A select-one type constraint can be easily encoded by them. For example, a constraint that variable x_2 has one and only one value of its domain, that is $X2 \wedge C2$, can be encoded by

$$x_{2:c} + x_{2:d} + x_{2:e} == 0 \qquad (4)$$

The final issue is that the interface between BDD and TMS should be compatible with existing systems, since many applications are developed and it costs very expensive to re-implement them from scratch.

4 Design of BMTMS

In this section, the *BDD-based Multiple-context TMS (BMTMS)*, is proposed to cope with the above issues. The whole system is depicted in Figure 4. BMTMS provides three interfaces to various kinds of existing systems:

1) interface to ATMS/CMS (Ⓐin Figure 4),
2) interface to extensions of ATMS such as consumer architecture [deK86b] (Ⓑin Figure 4), and
3) interface to inference engine (Ⓒin Figure 4).

These interfaces are only conceptual and thus treated uniformly within the BMTMS, since it can manipulate any logical formulas.

4.1 Interface functions

The criteria of designing the primitive functions is whether they can be implemented efficiently with BDDs.

1) **BDD primitive functions**:
`bddand`, `bddor`, `bddnot`, `bddrstr0`, `bddrstr1`, $\cdots$

Each function corresponds to a primitive operation of BDDs, and take BDDs as its arguments. For example, `(bddrstr0` BDD_f x`)` computes a BDD restricted to the case that x has value of 0, that is, $f_{|x=0}$.

2) **Variable ordering**:
   `variable-order` specifies the variable order.
3) **Logical operations**:
   `land`, `lor`, `lxor`, `imply`, `lnot`, $\cdots$
   These functions accept any number of arguments.
4) **Arithmetic logical operations**:
   `l*` (multiplication), `l+` (addition), `l/` (quotient), `l%` (remainder), `l-` (subtraction), `l<<` (left-shift) `l==` (equal to), ...
   These operations are implemented by simulating arithmetic logic unit (ALU) operations [Min93a]. Equation (4) is encoded by
   `(l== (l+` $x_{2:c}$ $x_{2:d}$ $x_{2:e}$`) 0)`.
5) **BDD graph operations**:
   `forall`, `exists`, `1-path`, $\cdots$
   `forall` is universal quantifier and `exists` is existential quantifier. Consider a function, f, is expressed in Shannon expansion,
   $$f = (\neg x \wedge f_{|x=0}) \vee (x \wedge f_{|x=1}).$$
   Universal quantifier $\forall x f$ is computed by
   $$\forall x f = f_{|x=0} \wedge f_{|x=1}.$$
   Or `(bddand (bddrstr0` f x`) (bddrstr1` f x`))`. Existential quantifier $\exists x f$ is computed by
   $$\exists x f = f_{|x=0} \vee f_{|x=1}.$$
   Or `(bddor (bddrstr0` f x`) (bddrstr1` f x`))`.
   `1-path` enumerates all the paths from the root of a BDD to the leaf of 1.
6) **Output functions**:
   `sop` returns a sum of products, which is a set of irredundant prime implicants. And `pos` returns a product of sums, which is a set of irredundant prime implicates.
7) **User defined data types**:
   In addition to system-defined binary and integer data, the user can define any data type by `defentity`.
   ```
   (defentity  datatype
        (slot1  slot2  ··· )
        (:exclusive-p) )
   ```
   A slot can be accessed by a function *datatype-slotname*. If `:exclusive-p` is specified, elements of a data defined by `defentity` hold exclusively. An operation for a data type is defined as follows:
   ```
   (defop (operation  datatype)
        argument-list  .  body )
   ```
   Its name is *datatype-operation*.
8) **Select function**:
   `(choice-of p`$_1 \cdots$ `p`$_n$`)` selects one element exclusively from the set. This function can be encoded by `(lor p`$_1 \cdots$ `p`$_n$`)` and `(l== (l+ p`$_1 \cdots$ `p`$_n$`) 1)`.
9) **Addition of Constraints**
   `(add-constraint` l C`)` adds a constraint C to the logical formula l. This is conceptually the same as `(setq` C `(land` C l`))`.

4.2 TMS Data Representation

Four kinds of nodes in TMS are encoded as follows:

1) **Premise:** the variable itself.
2) **Contradiction**: This node is used to represent a nogood relation in ATMS. It is not needed in the BMTMS, since a nogood relation can be represented by a logical formula.
3) **Assumption**: An assumption variable is introduced to express environments in the BMTMS, but is not discriminated from other nodes in BDDs.
4) **Normal node**: a variable itself.

Constraints between nodes are represented straightforward as follows:

- **Justification**, $n_1, \cdots, n_k \rightarrow c$, (c is an arbitrary clause), is encoded by
  `(imply (land n`$_1 \cdots$ `n`$_k$`) c)`.
- **nogood**{n$_1$,$\cdots$,n$_k$} is encoded by
  `(lnot (land n`$_1 \cdots$ `n`$_k$`))`.
- **class**{n$_1$, ..., n_k}, that selects one element from a set of nodes is encoded as follows:
  `(choice-of n`$_1 \cdots$ `n`$_k$`)`.

4.3 TMS operations

Let Π be a logical AND of all justifications and d be a node. An environment E is expressed by an AND of assumption variables, $a_1 \wedge ... \wedge a_k$. Implementations of some main TMS operations are listed below:

1) **Check whether all justification are not consistent.**
   If the BDD for Π reduces to 0, all justifications are not consistent.
2) **Check whether node d is consistent with environment E**: `node-consistent-with(`d, E`)`
   If the BDD for $\Pi \wedge d \wedge a_1 \wedge \cdots \wedge a_k$ reduces to 0, node d is not consistent with environment E.

3) **Compute the label of node d:** `tms-node-label`(d).

   Let $a_1, a_2, ...$ be assumption variables, and $d_1, d_2, ...$ be normal variables. The label of d is computed as follows:

   (1) Construct the BDD for $\Gamma = (\forall d_1, d_2, ...(\Pi \supset d))$. (2) Enumerate all prime implicates of Γ that contain d. This enumeration is described in Section 4.4. (3) Compute $\{X | X \supset d$ of a prime implicate$\}$, and this set is the label of d.

   If the justifications are restricted to Horn clauses (as in ATMS), the step (2) can be replaced with `pos`(Γ), which runs much faster.

4) **Check the status of node d:** `in-node?`, `out-node?`, `true-node?`, `false-node?`

   Let $a_1, a_2, ...$ be assumption variables, and $d_1, d_2, ...$ be normal variables. Construct the BDD for $(\Pi \supset d)$. If the BDD reduces 1 or 0, the node status of d is `:TRUE` and `:FALSE`, respectively. Otherwise, construct the BDD for $\forall d_1, d_2, ...(\Pi \supset d)$. If the BDD is not 0, the node status of d is `:IN`. Otherwise it is `:OUT`. (In addition, if the node status of $\neg d$ is `:OUT`, the node status of d is `:UNKNOWN`.)

5) **Force an assumption true or false.**

   These functions are implemented by `retract-assumption`(a_i or $\neg a_i$), which are encoded as follows: `(add-constraint` a_i or $\neg a_i$ Π`)`.

4.4 Prime implicates

In the research of boolean and switching functions, a *prime implicant* which is in the form of product of literals is usually used. (A literal is either a symbol or a negated symbol). Efficient algorithms for obtaining all prime implicants are proposed [CM92].

On the other hands, in truth maintenance, constraint satisfaction and logic programming, a *prime implicate* which is in the form of sum of literals is usually used. Since a conjunctive normal form is dual to a disjunctive normal form, a set of prime implicates can be computed by performing the duality operation on a set of prime implicants.

In other words, given a logical function $f(\boldsymbol{x})$, let $\bar{f}(\bar{\boldsymbol{x}})$ be a function gotten by exchanging $\vee$ with $\wedge$ and vice versa, and by replacing every literal with its negation, simultaneously. Then all the prime implicates for $\bar{f}(\bar{\boldsymbol{x}})$ are computed. Finally, exchanging $\wedge$ with $\vee$ and replacing every literal with its negation simultaneously computes every prime implicate.

The resulting set of prime implicates is enough for computing the label of a node, although it is not complete in the sense that any tautology such that $(y \vee \neg y)$ is a prime implicate,

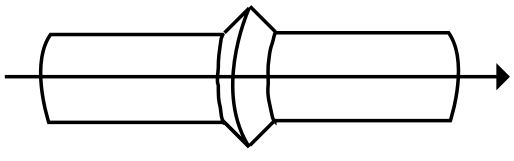


FIGURE 5 Two-pipe system ([FdK93, p.466])

In the BMTMS, all the prime implicates are computed directly from BDDs. The key point of the algorithm is that a BDD can be interpreted as another expansion on f.

$$f = (x_k \vee f_{|_{x_k=0}}) \wedge (\neg x_k \vee f_{|_{x_k=1}}),$$

The prime implicates of the function f can be obtained by combining the prime implicates of two subfunctions $f_{|_{x_k=0}}$ and $f_{|_{x_k=1}}$. The algorithm recursively computes prime implicates. This algorithm is a dual version of Coudert's [CM92]. Intermediate sets are maintained by Zero-Suppressed BDDs [CMF93], which are well suited for the set representation [Min93b].

4.5 Label enumeration

In the BMTMS, it has been shown that all TMS operations concerning labels except `tms-node-label` can be implemented without enumerating prime implicates. Although such enumeration is needed at least by abduction reasoning or dependency-directed search [RdK87], we guess that it is not so often in other applications. If this is true, many applications can be implemented by BDDs very efficiently.

4.6 Constraint and variable ordering

The constraint ordering algorithm used in the BMTMS is simpler than CCVO. The BMTMS does not construct a BDD for class constraints immediately when they are given to the system. Instead, adding those to the system is delayed until some variable of the class is used by some constraint. Variable ordering is determined the first time when it is used by such a constraint similar to that of CCVO.

5 APPLICATIONS OF BMTMS

In this section, some capabilities of the BMTMS are demonstrated in qualitative simulation.

Consider the system that has two pipes, A and C, which are connected by a joint, B (Fig.5). The pressure and flow in the system can be modeled by the following qualitative equations [deK91]:

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

where $[dx]$ denotes the sign (+, 0, -) of $\frac{dx}{dt}$.

First, the sign data type and its operations are defined as follows:

```
(defentity sign
   (positive zero negative)
   (:exclusive-p t) )

(defop (- sign) (s)
   (make-sign (sign-minus s)
       (sign-zero s) (sign-plus s) ))

(defop (+ sign) (s1 s2)
 (let ((xp (sign-plus s1))
       (x0 (sign-zero s1))
       (xm (sign-minus s1))
       (yp (sign-plus s2))
       (y0 (sign-zero s2))
       (ym (sign-minus s2)) )
 (make-sign
  (bddnot
   (lor (bddand x0 ym) (bddand x0 y0)
      (bddand xm y0) (bddand xm ym)))
   (bddnot
    (lor (bddand xp yp) (bddand xp y0)
      (bddand x0 yp) (bddand x0 ym)
      (bddand xm y0) (bddand xm ym)))
   (bddnot
    (lor (bddand x0 yp) (bddand x0 y0)
      (bddand xp yp) (bddand xp y0))))))

(defop (sign =0) (s) (sign-zero s))
```

Since the sign is defined with `:exclusive-p`, the operation =0 is specified quite simply instead of the following complicated definition:

```
(land (lnot (sign-plus s))
      (sign-zero s)
      (lnot (sign-minus s)) )
```

The logical formula that express $(x + y = 0)$ is defined as follows: (we use logical formula for economy of space.)

$$(y_m \wedge \neg y_p \wedge \neg y_0 \wedge \neg x_m \wedge x_p \wedge \neg x_0) \vee$$
$$(\neg y_m \wedge y_p \wedge \neg y_0 \wedge x_m \wedge \neg x_p \wedge \neg x_0) \vee$$
$$(\neg y_m \wedge \neg y_p \wedge y_0 \wedge \neg x_m \wedge \neg x_p \wedge x_0)$$

where indexes, $p, 0, m$, indicate three slots of the sign data type, respectively.

Let `C` be a Lisp variable that holds the constraint set. The above qualitative equations are encoded as follows:

```
(setq Pa (make-sign) Pb (make-sign)
      Pc (make-sign) Qab (make-sign)
      Qbc (make-sign) )
(add-constraint
    (sign-= (sign-- Pa Pb) Qab) C)
(add-constraint
    (sign-= (sign-- Pb Pc) Qbc) C)
(add-constraint
    (sign-= Qab Qbc) C)
```

Now, consider the situation that the pressure at A is increasing and one at C is not changing. This fact is encoded as follow:

```
(setq Constraint
   (land Constraint
         (sign-plus Pa)
         (sign-zero Pc) ))

(add-constraint Constraint C).
```

Finally, the result shows that the BDDs for
`(imply C (sign-plus Qab))`, and
`(imply C (sign-plus Qbc))`
reduce to 1. This means that both $[dQ_{AB}]$ and $[dQ_{BC}]$ are +, that is, the pressures at the interfaces between A and B, and between B and C are increasing. Note that this computation does not need enumerating prime implicates.

On the other hand, in QPE, every qualitative equation is expanded to a set of clausal form and all the prohibited combination are enumerated [deK91]. Then, all the prime implicates are computed. The final step is to apply the BCP to the set of all the prime implicates, which computes the labels of $[dQ_{AB}]$ and $[dQ_{BC}]$.

6 EVALUATION OF BMTMS

The current BMTMS system is implemented in Lisp and has two BDD packages (see Figure 4). One BDD package is implemented in C, which is called Boolean Expression Manipulator, BEM-II [Min93a]. This implementation of the BMTMS is referred to *BMTMS with BDD in C.* Another BDD package is written in Lisp and this implementation of the BMTMS is referred to *BMTMS with BDD in Lisp.* The timing data is measured on the SPARCStation10 with 128 MBytes of main memory.

6.1 Evaluating interface Ⓐ

N-Queens problem Two implementations of the BMTMS are compared with the ATMS which is implemented in Lisp. Two kinds of N-Queen problem programs by ATMS are used; one is by label update, and the other is by interpretation construction. Both programs computes column by column. The timing results of N-Queen problem programs are shown in Figure 6. The BMTMS with BDD in Lisp is much

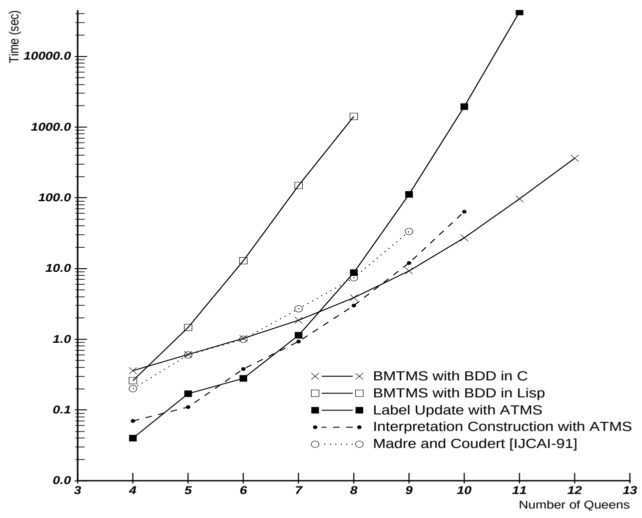


FIGURE 6 N-Queen problem results

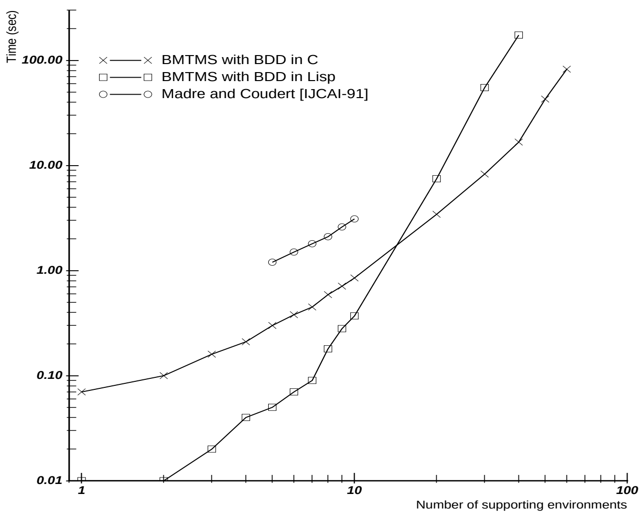


FIGURE 7 Minimal Cover results

slower than the ATMS. The reasons are twofold. (1) The algorithm of N-Queens is different. If the most fair algorithm by ATMS, which creates only one goal, installs all the justifications that justify the goal, and then computes labels, is used, it runs much slower than the BMTMS with BDD in Lisp and can get solutions only up to 6-Queen problem [Oku90]. (2) The BDD is not considered suitable for Lisp due to Lisp's memory management. Usually efficient implementations of BDD use only main memory, while such memory management is difficult in Lisp systems [Bry92; MIY91]. The BMTMS with BDD in C shows a good performance for large n.

Minimal cover This benchmark was used by Madre and Coudert [MC91]. The timing results of minimal cover are shown in Figure 7. The figure shows that the overheads of the BMTMS with BDD in Lisp exceed those of the BMTMS with BDD in C. This is caused by the overheads of Lisp runtime system.

6.2 Evaluating interface Ⓑ: ATMS trace file

We use an ATMS trace file generated by consumer architecture [deK86b], for which de Kleer's ATMS could not compute the labels. Both BMTMS implementations succeed in constructing BDDs by the constraint ordering algorithm. However, they fail in constructing BDDs due to combinatorial explosions, when the constraint and variable ordering mechanism is not used. The ordering algorithm is very simple, but proved effective.

6.3 Evaluating interface Ⓒ: Qualitative simulation

As described in the previous section, qualitative simulation of the two pipe system can be effectively computed by the BMTMS without enumerating all the prime implicates. However, this is a preliminary result and full assessment should be done by implementing the full set of QPE by the BMTMS.

7 CONCLUSIONS

In this paper, a new multiple-context truth maintenance system called BMTMS is proposed and its design and implementation is presented. The key idea of the BMTMS is to use BDDs to avoid enumerating prime implicates in manipulating general clauses. The two issues in applying BDDs to multiple-context TMSs are pointed out; variable ordering, and constraint ordering. To cope with these issues, the BMTMS schedules the constraint ordering by considering the dependency of variables and justifications (or constraints). The BMTMS also provides three level interface to existing problem solving and TMS systems so that BDD can be used in "plug-and-play" manner. Since the logical relations are stored in BDDs and most TMS operations can be performed without enumerating all prime implicates, the BMTMS runs efficiently. The BMTMS enumerates all prime implicates directly from BDDs. In addition, the capability of defining a data type and its operations makes it easy to implement applications with the BMTMS.

Future work includes implementation of the full QPE system with the BMTMS to demonstrate it power and open a new load to intelligent systems. Applying the BMTMS to various sophisticated expert systems is also an interesting area.

Acknowledgments

The authors thank Mr. Yuji Kukimoto of UCB, Dr. Shin'ichi Minato of NTT LSI Laboratories, and Mr. Hideki Isozaki of NTT Basic Research Laboratories for their discussions. The first author also thanks Dr. Ken'ichiro Ishii and Dr. Norihiro Hagita of NTT Basic Research Laboratories for their supports.

References

[Ake78] S.B., Akers. Binary Decision Diagrams. *IEEE Transactions on Computer*, Vol. C-27, No. 6, pages 509–516, 1978.

[Bry86] R.E. Bryant. Graph-based algorithm for Boolean function manipulation. *IEEE Transactions on Computer*, Vol. C-35, No. 5, pages 677–691, 1986.

[Bry92] R.E. Bryant. Symbolic Boolean Manipulations with Ordered Binary Decision Diagrams. *Computing Surveys*, Vol. 24, No. 3, pages 293–318, ACM, 1992.

[CM92] O. Coudert, and J.C. Madre. Implicit and Incremental Computation of Primes and Essential Primes of Boolean Functions. In *Proceedings of the 29th Design Automation Conference (DAC)*, pages 36–39, ACM/IEEE, 1992.

[CMF93] O. Coudert, J.C. Madre, and H. Fraisse. A New Viewpoint on Two-Level Logic Minimization. In *Proceedings of the 30th Design Automation Conference (DAC)*, pages 625–630, ACM/IEEE, 1993.

[deK86a] J. de Kleer. An Assumption-based TMS. *Artificial Intelligence*, **28**:127-162, 1986.

[deK86b] J. de Kleer. Extending the ATMS. *Artificial Intelligence*, **28**:163-196, 1986.

[deK89] J. de Kleer. A Comparison of ATMS and CSP Techniques. In *Proceedings of the Eleventh Internatioal Joint Conference on Artificial Intelligence (IJCAI)*, pages 290–296, 1989.

[deK91] J. de Kleer. Compiling Devices: Locality in a TMS. In Faltings, B. and Strauss, P. (eds): *Recent Advances in Qualitative Physics*, MIT Press, 1991.

[Dow84] W.F. Dowling and J.H. Gallier. Linear time algorithms for testing the satisfiability of propositional horn formulas. *Journal of Logic Programming,* vol.3, pages 267–284, 1984.

[FdK93] K. Forbus, and J. de Kleer. *Building Problem Solvers*, MIT Press, 1993.

[LCM92] B. Lin, O. Coudert, and J.C. Madre. Symbolic prime generation for muliple-valued functions. In *Proceedings of the 29th Design Automation Conference (DAC)*, paages 40–44, ACM/IEEE, 1992.

[MC91] J.C. Madre, and O. Coudert. A logically complete reasoning maintenance system. In *Proc. of the Twelfth Internatioal Joint Conference on Artificial Intelligence (IJCAI)*, pages 294–299, 1991.

[MIY91] S. Minato, N. Ishiura, and Y. Yajima. Shared Binary Decision Diagram with Attributed Edges for Efficient Boolean Function Manipulation. In *Proceedings of the 27th Design Automation Conference (DAC)*, pages 52–57, IEEE/ACM, 1990.

[Min93a] S. Minato. BEM-II: An arithmetic Boolean expression manipulator using BDDs. *IEICE Transactions of Fundamentals*, Vol. E76-A, No. 10, pages 1721–1729, 1993.

[Min93b] S. Minato. Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems. In *Proceedings of the 30th Design Automation Conference (DAC)*, pages 272–277, ACM/IEEE, 1993.

[Oku90] H.G. Okuno. AMI: A New Implementation of ATMS and its Parallel Processing (*in Japanese*). *Journal of Japanese Society for Artificial Intelligence*, Vol.5, No.3, pages 333–342, 1990.

[Oku94] H.G. Okuno. Reducing Combinatorial Explosions in Solving Search-Type Combinatorial Problems with Binary Decision Diagrams (*in Japanese*). *Transactions of Information Processing Society of Japan* Vol.35, No.5, pages 739–753, 1994.

[OST96] H.G. Okuno, O. Shimokuni, and H. Tanaka. Binary Decision Diagram based Multipli-Context Truth Maintenance System BMTMS (*in Japanese*). *Journal of Japanese Society for Artificial Intelligence*, Vol.10, No.2, 1996.

[RdK87] R. Reiter, and J. de Kleer. Foundations of Assumption-Based Truth Maintenance System. In *Proceedings of National Conference on Artificial Intelligence (AAAI)*, pages 183–188, 1987.