Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Zero-Suppressed BDDs and Their Applications

Shin-ichi Minato

NTT Network Innovation Laboratories, Yokosuka, 239-0847 Japan.
E-mail: minato@exa.onlab.ntt.co.jp, minato@ieee.org

Received: date / Revised version: date

Abstract. In many real-life problems, we are often faced with manipulating *sets of combinations*. In this article, we study a special type of OBDDs, called *Zero-suppressed BDD (ZBDD)*. This data structure represents sets of combinations more efficiently than using original OBDDs. We discuss the basic data structures and algorithms for manipulating ZBDDs in contrast with the original OBDDs. We also present some practical applications of ZBDDs, such as solving combinatorial problems with unate cube set algebra, logic synthesis method, Petri net processing, etc. We show that ZBDD is a useful option in the OBDD techniques, suitable for a part of practical applications.

1 Introduction

Ordered Binary Decision Diagrams (OBDDs), which are reviewed in the overview article[9] of this special section, are powerful means for computer processing of Boolean functions. In many cases, this data structure requires smaller memory space for storing Boolean functions and calculates values of functions faster than with truth tables or logic expressions.

As our understanding of OBDDs has deepened, their range of applications has broadened. In VLSI CAD problems, we are often faced with manipulating not only Boolean functions but also *sets of combinations*. By mapping the data into Boolean space, a set of combinations can be represented in an OBDD. This method enables us to implicitly process a huge number of combinations, which has never been practical before. For instance, Coudert, et al.[4,5] presented a fast method for generating a huge number of prime implicant sets involved in a Boolean function, which is one of the important results in two-level logic minimization problems. For another instance, Lin, et al.[12] also used the OBDD-based techniques for manipulating sets of combinations in solving general covering problems.

Those implicit manipulation method is more efficient than with the conventional data structures, however, there is still room to improve the efficiency because OBDDs were originally designed for representing Boolean functions, not completely fit to the sets of combinations.

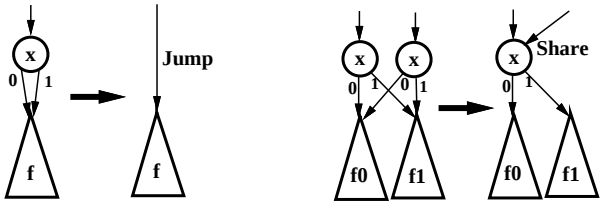
In this article, we study a special type of OBDDs for manipulating sets of combinations. This data structure, called *Zero-suppressed BDD (ZBDD)*[16], represents sets of combinations more efficiently than using original OBDDs. In the following sections, we discuss the basic data structures and algorithms for manipulating ZBDDs in contrast with the original OBDDs. We then present some practical applications of ZBDDs.

This article is structured as follows: In Section 2, first we review the way to represent sets of combinations using original OBDDs. We then describe the data structures of ZBDDs, followed by the basic algorithms for manipulating them. We discuss the difference in manipulation methods of ZBDDs and original OBDDs, and show the relationship between ZBDDs and OFDDs.

In Section 3, we discuss *unate cube set algebra*, which is useful for manipulating sets of combinations. We show that the operations of the algebra are efficiently implemented by using ZBDDs. Several applications of unate cube set manipulation are presented.

In Section 4, we present the ZBDD-based logic synthesis method, which is the most successful application of ZBDDs. This method is based on *binate cube set algebra*, which is different from the unate one. We show the fast method for generating compact cube set representation for given Boolean functions. Those cube sets can be factored into multi-level logic networks by using a fast division algorithms.

In Section 5, we survey some other published works of ZBDD applications, such as processing of Petri nets,



(a) Node deletion. (b) Node sharing.

Fig. 1. Reduction rules of original OBDDs.

$F(abc)$:

abc	F
000	0
100	0
010	0
110	1
001	0
101	0
011	0
111	1

$F(abcd)$:

abcd	F
0000	0
1000	0
0100	0
1100	1
0010	0
1010	0
0110	0
1110	1
0001	0
1001	0
0101	0
1101	1
0011	0
1011	0
0111	0
1111	1

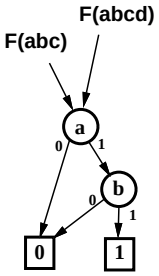


Fig. 2. OBDDs for Boolean functions.

Graph optimization problems, and manipulating polynomial expressions.

Finally, the conclusions of this article are stated in Section 6.

2 Zero-Suppressed BDDs

In this section, we describe the basic data structures and algorithms of Zero-Suppressed BDDs for processing sets of combinations, and compare them to ordinary OBDDs as well as OFDDs.

2.1 Reduction Rules of OBDDs

Here we review the reduction rules of original OBDDs, and then show a problem which motivates us to develop a new type of OBDDs. As mentioned in the overview article[9], OBDDs are based on the following two reduction rules:

- *S-deletion rule*: Delete all redundant nodes whose two edges point to the same node. (Fig. 1(a))
- *merging rule*: Share all equivalent subgraphs. (Fig. 1(b))

We know that the reduction rules make OBDDs compact and canonical for Boolean functions under a fixed variable ordering.

One may ask which reduction rule has more significant effects to make OBDDs compact? Liaw et al.[11] and Wegener[30] show that, for general (or random) Boolean functions, the merging rule makes a much more significant contribution to storage saving than the deletion rule. However, here we show that the deletion rule is also important in some practical environment. For example, Fig. 2 illustrates the Boolean functions of $(a \wedge b)$ in different input domains $F(abc)$ and $F(abcd)$. They require the different representations in using truth tables, however, those OBDDs become the same form with only two decision nodes. In other words, OBDDs are independent of the input domains as long as the logic expressions are the same. The irrelevant variables are automatically suppressed by the node deletion rule. This property is very important in some practical applications with a large number of variables.

2.2 OBDDs for Sets of Combinations

Presently, there have been many works on OBDD applications. As mentioned in previous section, some kind of applications use OBDDs not for simply representing Boolean functions, but for processing *sets of combinations* (e. g. [4,5,12]). Sets of combinations often appear in solving combinatorial problems. The representation and manipulation of sets of combination are important techniques for many applications.

A combination of n items can be represented by an n-bit binary vector, $(x_1 x_2 \ldots x_n)$, where each bit, $x_k \in \{1, 0\}$, expresses whether or not the item is included in the combination. A set of combinations can be represented by a list of the combination vectors. In other words, a set of combinations is a subset of the power set of n items.

A set of combinations can be mapped into Boolean space by using n-input variables for each bit of the combination vector. If we choose any one combination vector, a Boolean function determines whether the combination is included in the set of combinations. Such Boolean functions are called *characteristic functions*. The set operations such as union, intersection, and difference can be performed by logic operations on characteristic functions.

By using OBDDs for characteristic functions, we can manipulate sets of combinations efficiently. Due to the effect of node sharing, OBDDs compactly represent sets of huge number of combinations. OBDDs can be generated and manipulated within a time roughly proportional to the size of graphs, whereas the previous data structures (such as arrays and sequential lists) require a time proportional to the number of combinations.

Despite the efficiency of OBDDs, there is one inconvenience. For example, $S(abc)$ and $S(abcd)$ shown in Fig. 3 represent the same set of combinations $\{a, b\}$ if we ignore the irrelevant input variables c and d. In this case, the OBDDs for $S(abc)$ and $S(abcd)$ are not identical. Here we can see that the forms of OBDDs depend on the input domains when representing sets of combinations. This inconvenience comes from the difference

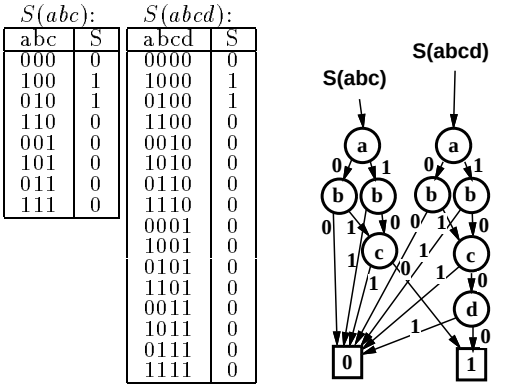


Fig. 3. OBDDs for sets of combinations.

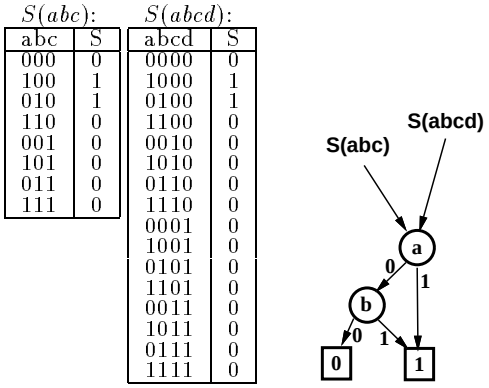


Fig. 5. ZBDDs for sets of combinations.

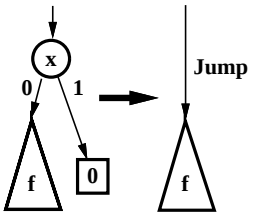


Fig. 4. New reduction rule for ZBDDs.

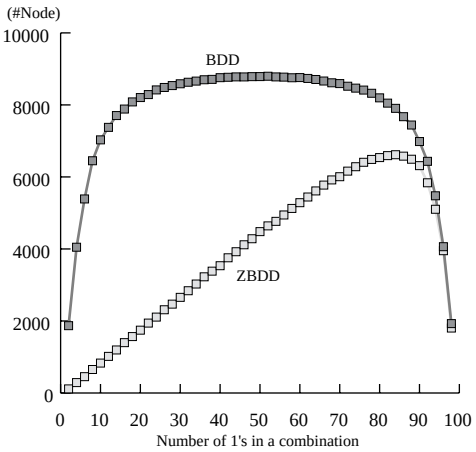


Fig. 6. Comparison of ZBDDs and OBDDs.

in the model of default behavior. For sets of combinations, we assume that irrelevant items never appear in any combination, so those input variables must be zero to satisfy the characteristic functions. Unfortunately, the OBDD node deletion rule is not effective to such irrelevant variables, and many useless nodes are generated when we manipulate sparse combinations. This is the reason why we need another type of OBDDs for manipulating sets of combinations.

2.3 Zero-Suppressed Reduction Rule

For representing sets of combinations efficiently, Minato[16] introduced the following node deletion rule:

– Delete all nodes whose 1-edge points to the 0-terminal node, and then connect the edges to the other subgraph directly as shown in Fig. 4.

This is also called *pD-deletion rule*[9]. Notice that we do not delete the nodes whose two edges point to the same node, which used to be deleted by the original rule. The zero-suppressed deletion rule is asymmetric for the two edges, as we do not delete the nodes whose 0-edge points to a 0-terminal node. We call this type of OBDDs *Zero-suppressed BDDs (ZBDDs).*

Theorem 1. *If the input domain and variable ordering are fixed, a ZBDD uniquely represents a Boolean function.*

(Proof): See Appendix.

Figure 5 shows the ZBDDs for the same sets of combinations shown in Fig. 3. The form of ZBDDs are independent of the input domain. The ZBDD node deletion rule automatically suppresses the variables which never appear in any combination. This feature is important when we manipulate *sparse* combinations.

For evaluating the ZBDD node deletion, we conducted a statistical experiment. We generated one hundred pieces of 100-bit vectors, each of which randomly selects k-bit out of 100-bit. We then compared the sizes of the ZBDDs and original OBDDs for the set of the hundred random combinations[1]. The result in Fig. 6 shows that ZBDDs are much more compact than original ones — especially when k is small. This means that ZBDDs are particularly effective for representing sets of sparse combinations. The effect weakens for large k; however, we can use complement combination vectors to make k less than 50%. For example, the combination selecting 90-bit out of 100 is equivalent to selecting the other 10-bit.

Another advantage of ZBDDs is that each path from the root node to the 1-terminal node, which we call *1-*

[1] We tried many times with different random seeds and show the average.

path, corresponds to each combination vector in the set. In other words, the number of 1-paths in a ZBDD is exactly equal to the number of combinations in the set. This beautiful property is broken if we use the original OBDD reduction rules.

From the above observations, we can conclude that ZBDDs are more suitable for representing sets of combinations than original OBDDs are. On the other hand, original OBDDs will be superior to ZBDDs when manipulating ordinary Boolean functions. The difference is in the models of default behaviors: "must be zero" in sets of combinations, or "both the same" in Boolean functions. We may choose one of the two types of OBDDs according to the nature of applications.

2.4 Basic Operations of ZBDDs

In this section, we describe the algorithms of ZBDD manipulation. As well as original OBDDs for Boolean functions, ZBDDs support many operations for sets of combinations. Here we show the most of basic operations. In the following, P and Q indicate instances of sets of combinations represented by ZBDDs, and v means an input variables.

"$\emptyset$"	the empty set. (0-terminal node.)
"**1**"	the set of only the null combination $\{00\ldots0\}$. (1-terminal node.)
P.top	the variable at the root of the ZBDD.
P.offset(v)	selects the subset of the combinations each of which does not include v.
P.onset(v)	selects the subset of the combinations including v, and then delete v from each combination.
P.change(v)	inverts v on each combination. (swaps onset and offset of P.)
$P \cup Q$	the union set of P and Q.
$P \cap Q$	the intersection set of P and Q.
$P - Q$	the difference set. (combinations in P but not in Q.)
P.count	the number of combinations in P.

Figure 7 shows an example of constructing ZBDDs. Any set of combinations can be generated by a sequence of those basic operations. Since ZBDDs give canonical forms for sets of combinations, equivalence checking is quite easily performed. Notice that we do not support the unary complement operation, which is essential in OBDDs. Its absence here is reasonable, as we need to define a universal set U to compute complement set $\overline{P}$. We use the difference operation $(U - P)$ to compute it.

We show here that those ZBDD operations can be executed recursively, like the ones for original OBDDs. First, to describe the algorithms simply, we define the procedure Getnode(v, P_0, P_1), which generates (or makes a reference to) a node with the variable v and two subgraphs P_0, P_1. The algorithm is shown in Fig. 8. In this

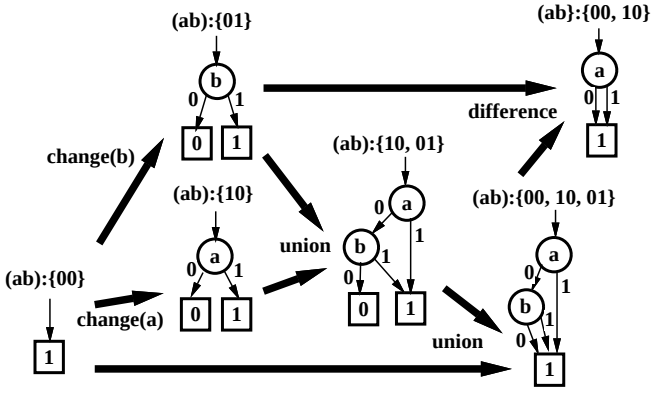


Fig. 7. Construction of ZBDDs for sets of combinations.

```
Getnode (v, P0, P1) {
    if (P1 == ∅) return P0;  /* node deletion */
    P = search a node with (v, P0, P1) in uniq-table;
    if (P exist) return P;   /* node sharing */
    P = generate a node with (v, P0, P1);
    append P to the uniq-table;
    return P;
}
```

Fig. 8. Algorithm of "Getnode".

procedure, we use a hash table, called *uniq-table*, to keep each node unique. The node deletion and sharing process is encapsulated in the Getnode procedure.

The algorithms of the ZBDD operations are shown in Fig. 9. In the description, P.top $< Q$.top means the root node of P is higher (farther from the terminal nodes) than the root of Q. P_0 and P_1 means the two subgraphs of the root node of P.

In these algorithms, we use a cache which stores the results of recent operations, similarly to the OBDD operations. By referring to the cache before every recursive call, we can avoid duplicate executions for equivalent subgraphs. This enable us to execute these operations in a time that is roughly proportional to the size of the graphs.

2.5 Using Attributes in ZBDD Edges

In many OBDD packages, the *complement edges* are commonly used to reduce the computation time and memory requirement. Minato[16] presented a similar technique for ZBDDs, called *0-element edge*, but the function of the attribute is slightly different from conventional ones. This attribute indicates that the pointing subgraph includes the null combination vector "$00\ldots0$" in the set of combinations.

Similarly to the complement edges, we have to place a couple of constraints in using 0-element edges to keep the uniqueness of the graphs:

- Use the 0-terminal only.

```
P.offset(v) {
  if (P.top = v) return P_0;
  if (P.top > v) return P;
  R ← cache[P.offset(v)]; if(R exists) return R;
  R ← Getnode(P.top, P_0.offset(v), P_1.offset(v));
  cache[P.offset(v)] ← R;
  return R;
}
P.onset(v) {
  if (P.top = v) return P_1;
  if (P.top > v) return ∅;
  R ← cache[P.onset(v)]; if(R exists) return R;
  R ← Getnode(P.top, P_0.onset(v), P_1.onset(v));
  cache[P.onset(v)] ← R;
  return R;
}
P.change(v) {
  if (P.top = v) return Getnode(v, P_1, P_0);
  if (P.top > v) return Getnode(v, ∅, P);
  R ← cache[P.change(v)]; if(R exists) return R;
  R ← Getnode(P.top, P_0.change(v), P_1.change(v));
  cache[P.change(v)] ← R;
  return R;
}
procedure(P ∪ Q) {
  if (P = ∅) return Q;
  if (Q = ∅ or P = Q) return P;
  R ← cache[P ∪ Q]; if(R exists) return R;
  if (P.top < Q.top) R ← Getnode(P.top, P_0 ∪ Q, P_1);
  if (P.top > Q.top) R ← Getnode(Q.top, P ∪ Q_0, Q_1);
  if (P.top = Q.top) R ← Getnode(P.top, P_0 ∪ Q_0, P_1 ∪ Q_1);
  cache[P ∪ Q] ← R;
  return R;
}
procedure(P ∩ Q) {
  if (P = ∅ or Q = ∅) return ∅;
  if (P = Q) return P;
  R ← cache[P ∩ Q]; if(R exists) return R;
  if (P.top < Q.top) R ← P_0 ∩ Q;
  if (P.top > Q.top) R ← P ∩ Q_0;
  if (P.top = Q.top) R ← Getnode(P.top, P_0 ∩ Q_0, P_1 ∩ Q_1);
  cache[P ∩ Q] ← R;
  return R;
}
procedure(P − Q) {
  if (P = ∅ or P = Q) return ∅;
  if (Q = ∅) return P;
  R ← cache[P − Q]; if(R exists) return R;
  if (P.top < Q.top) R ← Getnode(P.top, P_0 − Q, P_1);
  if (P.top > Q.top) R ← P − Q_0;
  if (P.top = Q.top) R ← Getnode(P.top, P_0 − Q_0, P_1 − Q_1);
  cache[P − Q] ← R;
  return R;
}
P.count {
  if (P = ∅) return 0;
  if (P = 1) return 1;
  sum ← cache[P.count]; if(sum exists) return sum;
  sum ← P_0.count + P_1.count;
  cache[P.count] ← sum;
  return sum;
}
```

Fig. 9. Algorithm of the basic operations.

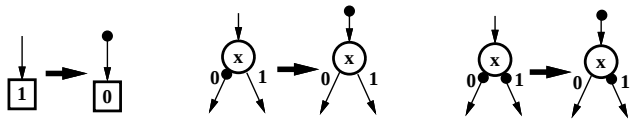


Fig. 10. Rules in using 0-element edges.

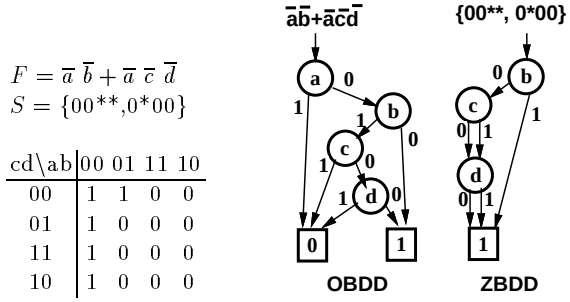


Fig. 11. Comparison of OBDD and ZBDD.

– Do not use 0-element edges at the 0-edge on each node.

If necessary, 0-element edges can be carried over as shown in Fig. 10. The constraint rules can also be encapsulated in the Getnode procedure.

0-element edges accelerate operations on ZBDDs. For example, the result of $(P \cup \{00\ldots0\})$ depends only on whether or not P includes the null combination. In such a case, we can get the result in a constant time when using 0-element edges; otherwise we have to repeat the expansion until P becomes a terminal node.

2.6 Comparison of OBDD and ZBDD Construction

Someone may easily consider that we can improve the performance of an OBDD application program just by replacing OBDDs by ZBDDs when the target OBDDs are reduced by using ZBDDs. However, if we construct ZBDDs in the same way as original OBDDs, we sometimes fail to have the benefit of ZBDDs. For example, suppose an OBDD and a ZBDD for the same Boolean function (or set of combinations) shown in Fig. 11. This function is expressed as $(\overline{a}\,\overline{b} + \overline{a}\,\overline{c}\,\overline{d})$, and its OBDD can be constructed in the way as Fig. 12. Since OBDDs and ZBDDs have one-to-one mapping, a ZBDD can be constructed in the same way as OBDDs, as shown in Fig. 13. In this example, we can see that the intermediate results of ZBDDs are not efficiently represented although the final ZBDD is more compact than the original one. For taking the full of benefit of ZBDDs, we had better use another sequence of operations as shown in Fig. 14. This example shows that the efficient operation sequences are different in using ordinary OBDDs and ZBDDs.

In general, ZBDDs are more efficient than original OBDDs when manipulating the sets of "sparse combinations" (it is different from "sparse sets" of combinations.)

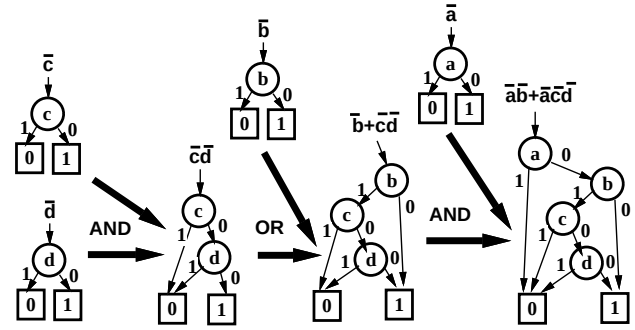


Fig. 12. Ordinary OBDD construction.

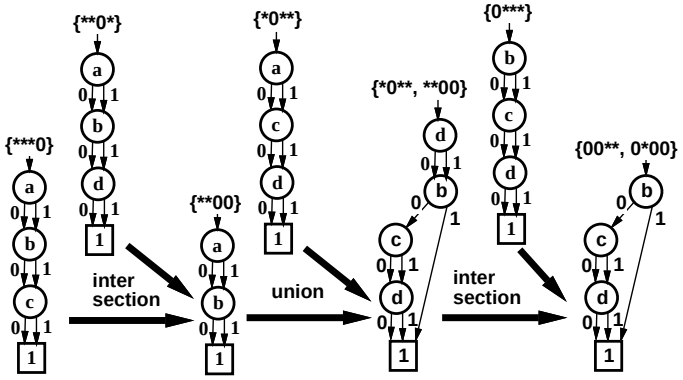


Fig. 13. ZBDD construction in the same way as OBDD.

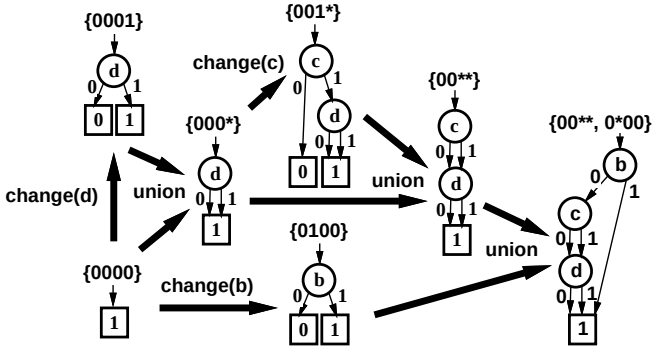


Fig. 14. Good ZBDD construction.

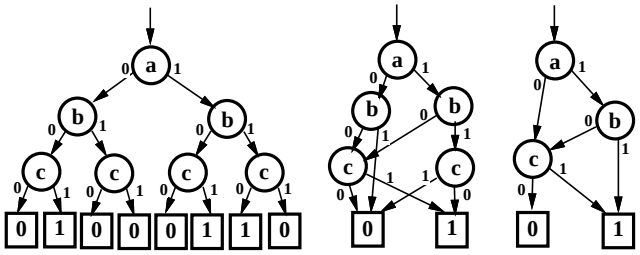
Anyway, the improvement of ZBDDs from OBDDs is at most n times (n: number of input variables) in terms of time and space. For some practical applications it is remarkably effective, but would be negligible in dealing with small n for exponentially large OBDDs.

2.7 Relationship between ZBDDs and OFDDs

As described in the overview article[9], there is another type of OBDD based on AND-EXOR logic operations. *OFDDs (Ordered Functional Decision Diagrams)*, presented by Kebschull et al.[10], are based on the following expansion:

$$f = f_{(x=0)} \oplus x \cdot (f_{(x=0)} \oplus f_{(x=1)}).$$

This expansion is called *(positive) Davio's expansion*. An OFDD is derived by reducing a binary tree graph, shown in Fig. 15(a). Each node of this binary tree represents Davio's expansion, i.e. the 0-edge points to $f_{(x=0)}$, and the 1-edge points to $(f_{(x=0)} \oplus f_{(x=1)})$.



(a) Non-reduced OFDD tree. (b) Ordinary reduction. (c) Zero-sup. reduction.

Fig. 15. OFDDs for $(ab \oplus ac \oplus c)$.

The first paper of OFDD used the same deletion rule as OBDDs, but later[8] it is understood that Zero-suppressed (pD-deletion) rule is more fit to OFDDs. For example, the binary tree is reduced to Fig. 15(b) by the ordinary reduction rules, and more reduced to Fig. 15(c) by the Zero-suppressed rule.

OFDD has another property that each 1-path (the path from the root node to a 1-terminal node) is corresponding to a product term in the AND-EXOR two-level expression, called *(positive polarity) Reed-Muller expression*. For example, the OFDD shown in Fig. 15 has three 1-paths: (abc):$\{110, 101, 001\}$. They corresponds to the product terms in the Reed-Muller expression: $(ab \oplus ac \oplus c)$. In other words, the OFDDs can also be regarded as the ZBDDs representing sets of product terms in the Reed-Muller expressions. This observation explains why the complement edge used in OFDDs[1] follows the same rule as the 0-element edge used in ZBDDs.

3 Unate Cube Set Algebra Based on ZBDDs

In this section, we discuss unate cube set algebra[17] based on ZBDD manipulation. A cube set consists of a number of cubes, each of which is a combination of literals. *Unate* cube sets allow us to use only positive literals, not the negative ones. We sometimes use cube sets to represent Boolean functions, but they are usually *binate* cube sets containing both positive and negative literals. Those binate cube sets are discussed in Section 4.

3.1 Basic Operations

Unate cube set expressions consist of trivial sets and algebraic operators as follows.

0	(empty set),
1	(unit set),
var	(single literal set),
$\cap$	(intersection),
$+$	(union),
$-$	(difference),
$*$	(product),
$/$	(quotient of division),
$\%$	(remainder of division).

The unit set "1" includes only a "null" cube which consists of no literal. A single literal set var includes only a cube which consists of one literal. In the rest of this section, a lowercase letter denotes a literal, and an uppercase letter denotes a cube set expression. We sometimes omit "$*$" for the simplicity.

Here are examples of calculating unate cube sets by those operations.

$$\begin{aligned}
\{ab,b,c\} \cap \{ab,1\} &= \{ab\} \\
\{ab,b,c\} + \{ab,1\} &= \{ab,b,c,1\} \\
\{ab,b,c\} - \{ab,1\} &= \{b,c\} \\
\{ab,b,c\} * \{ab,1\} &= (ab*ab) + (ab*1) + (b*ab) \\
&\quad + (b*1) + (c*ab) + (c*1) \\
&= \{ab,abc,b,c\}
\end{aligned}$$

The product operation "$*$" generates all possible concatenations of two cubes in respective cube sets.

The following equations are observed in the unate cube set calculation:

$$\begin{aligned}
&P + P = P, \quad P * 1 = P, \\
&a * a = a \quad (P * P \neq P \quad \text{in general}), \\
&P * Q = Q * P, \\
&(P * Q) * R = P * (Q * R), \\
&P * (Q + R) = (P * Q) + (P * R), \\
&P - Q = Q - P \quad \text{if and only if} \quad P = Q.
\end{aligned}$$

Dividing P by Q acts to seek out the two cube sets P/Q (quotient) and $P\%Q$ (remainder) under the equality:

$$P = Q * (P/Q) + (P\%Q).$$

In general this solution is not unique. Here we apply the following rules to fix the solution with reference to the *weak-division method*[2].

1. When Q includes only one cube, (P/Q) is obtained by extracting a subset of P, which consists of the cubes including all the literals in Q's cube, and then eliminating Q's literals. For example,
$$\{abc,bc,ac\}/\{bc\} = \{a,1\}.$$
2. When Q consists of multiple cubes, (P/Q) is the intersection of all the quotients dividing P by respective cubes in Q. For example,
$$\begin{aligned}
&\{abd,abe,abg,cd,ce,ch\}/\{ab,c\} \\
&= (\{abd,abe,abg,cd,ce,ch\}/\{ab\}) \\
&\quad \cap (\{abd,abe,abg,cd,ce,ch\}/\{c\}) \\
&= \{d,e,g\} \cap \{d,e,h\} \\
&= \{d,e\}.
\end{aligned}$$
3. $(P\%Q)$ can be obtained by calculating $P - Q*(P/Q)$.

In Section 2, we defined three operations — P.offset(v), P.onset(v), and P.change(v) — as the basic methods of manipulating ZBDDs. Now the three operations can be regarded as the special cases of the division operation in the unate cube set algebra. P.offset(v) can be described as $(P\%v)$, and P.onset(v) becomes (P/v). P.change(v) is also be written as $(P\%v)*v+(P/v)$.

3.2 Algorithms

We show here that the basic operations of unate cube set algebra can be efficiently calculated using ZBDDs. The empty set "0" corresponds to the 0-terminal, and the unit set "1" is the 1-terminal node. A single literal set x_k corresponds to the single-node graph pointing directly to the 0- and 1-terminal node. The intersection, union, and difference operations are the same as the basic ZBDD operations shown in Section 2. The other three operations — product, quotient, and remainder — are not included in the basic ones. If we calculate those operations by processing each cube one by one, the computation time will depend on the length of expressions. Such a procedure is impractical when we deal with very large numbers of cubes. Minato[17] presented the fast algorithms for computing them.

The algorithms are based on the divide-and-conquer method. We first explain the multiplication algorithm. When v is the highest one between P.top and Q.top, P and Q are then factored into the two parts:

$$P = v * P_1 + P_0, \quad Q = v * Q_1 + Q_0.$$

The product $(P*Q)$ can be expanded as:

$$(P*Q) = v*(P_1*Q_1 + P_1*Q_0 + P_0*Q_1) + P_0*Q_0.$$

Each sub-product term can be computed recursively. The expressions are eventually broken down into trivial ones and the results are obtained. By referring to the cache storing recent results of the operations, we can avoid duplicate executions for equivalent subgraphs. Consequently, the execution time depends on the size of ZBDDs rather than on the number of cubes and literals[2]. This algorithm is described in detail in Fig. 16.

Division is computed in the same recursive manner. Suppose $v = Q$.top (here we do not use P.top), and that $P_0, P_1, Q_0, and Q_1$ are the subsets of cubes factored by v. (Notice that $Q_1 \neq 0$, since v is chosen from Q.) Then the quotient (P/Q) can be described as

$$\begin{aligned}
&(P/Q) = (P_1/Q_1), \quad \text{when} \quad Q_0 = 0. \\
&(P/Q) = (P_1/Q_1) \ \cap \ (P_0/Q_0), \quad \text{otherwise.}
\end{aligned}$$

[2] We have not known the exact complexity of this algorithm, but empirically, about square of graph size in many cases.

```
procedure(P * Q)
{   if (P.top > Q.top) return (Q * P) ;
    if (Q = 0) return 0 ;
    if (Q = 1) return P ;
    R ← cache("P * Q") ; if (R exists) return R ;
    v ← P.top ; /* the highest variable in P */
    (P0, P1) ← factors of P by v ;
    (Q0, Q1) ← factors of Q by v ;
    R ← v (P1 * Q1 + P1 * Q0 + P0 * Q1) + P0 * Q0 ;
    cache("P * Q") ← R ;
    return R ;
}
```

Fig. 16. Algorithm for multiplication.

```
procedure(P/Q)
{   if (Q = 1) return P ;
    if (P = 0 or P = 1) return 0 ;
    if (P = Q) return 1 ;
    R ← cache("P/Q") ; if (R exists) return R ;
    v ← Q.top ; /* the highest variable in Q */
    (P0, P1) ← factors of P by v ;
    (Q0, Q1) ← factors of Q by v ; /* (Q1 ≠ 0) */
    R ← P1/Q1 ;
    if (R ≠ 0) if (Q0 ≠ 0) R ← R ∩ P0/Q0 ;
    cache("P/Q") ← R ;
    return R ;
}
```

Fig. 17. Algorithm for division.

Each sub-quotient term can be computed recursively. Whenever we find that one of the sub-quotients (P_1/Q_1) or (P_0/Q_0) results in 0, $(P/Q) = 0$ becomes obvious and we no longer need to compute it. Using the cache technique avoids duplicate executions for equivalent subgraphs. This algorithm is described in Fig. 17. The remainder $(P\%Q)$ can be determined by calculating $P - P * (P/Q)$.

3.3 Application of Unate Cube Set Calculation

In this section, we present several examples using unate cube set algebra for describing and solving combinatorial problems.

8-Queens Problem

First we show an application to the 8-Queens problem, which is a commonly known combinatorial problem. We define 64 input variables to represent the squares on a chessboard. Each variable denotes whether or not there is a queen on that square. The problem can be described with the variables as follows:

- In a particular column, only one variable is "1."
- In a particular row, only one variable is "1."
- On a particular diagonal line, one or no variable is "1."

Table 1. Results on N-queens problems.

N	Lit.	Sol.	OBDD nodes	ZBDD nodes	(B/Z)	(Z/S)
4	16	2	29	8	3.6	4.0
5	25	10	166	40	4.2	4.0
6	36	4	129	24	5.4	6.0
7	49	40	1098	186	5.9	4.65
8	64	92	2450	373	6.6	4.05
9	81	352	9556	1309	7.3	3.72
10	100	724	25944	3120	8.3	4.31
11	121	2680	94821	10503	9.0	3.92
12	144	14200	435169	45833	9.5	3.23
13	169	73712	2044393	204781	10.0	2.78

(B/Z): OBDD nodes / ZBDD nodes
(Z/S): ZBDD nodes / Sol.

We can describe this constraints by using logic expressions, and the problem can be solved by constructing OBDDs. (See the article[20] for ones interested in this topic.) The 8-Queens problem is one of the instances where ZBDDs are more efficient than using OBDDs.

As discussed in Section 2.6, the sequence of ZBDD construction should be different from OBDD's one. By using unate cube set calculation, the construction sequence is described in the following way.

$$
\begin{aligned}
S_1 &\leftarrow x_{11} + x_{12} + \ldots + x_{18} \\
S_2 &\leftarrow x_{21}(S_1 \% x_{11} \% x_{12}) + x_{22}(S_1 \% x_{11} \% x_{12} \% x_{13}) \\
&\quad + \ldots + x_{28}(S_1 \% x_{17} \% x_{18}) \\
S_3 &\leftarrow x_{31}(S_2 \% x_{11} \% x_{13} \% x_{21} \% x_{22}) \\
&\quad + x_{32}(S_2 \% x_{12} \% x_{14} \% x_{21} \% x_{22} \% x_{23}) \\
&\quad + \ldots + x_{38}(S_2 \% x_{16} \% x_{18} \% x_{27} \% x_{28}) \\
S_4 &\leftarrow \ldots
\end{aligned}
$$

The ZBDD is constructed in step by step from S_1 to S_8. Each step is the following meaning.

S_1: Represents all choices of putting a queen at the first row.
S_2: Represents all choices of putting a queen at the second row so as not to violate the first queen.
S_3: Represents all choices of putting a queen at the third row so as not to violate the first and second queens.
...
S_8: Represents all choices of putting the eighth queen so as not to violate the other queens.

The ZBDD for S_8 contains all possible solution of the 8-Queens problem.

Okuno[25] reported experimental results for N-Queens problems to compare the efficiencies of ZBDDs and conventional OBDDs. In Table 1, the column "Sol." shows the number of solutions. "OBDD nodes" and "ZBDD nodes" show the size of OBDDs using Boolean algebra and ZBDDs using unate cube set algebra, respectively. We can see that ZBDDs are about N times fewer than

OBDDs. The result of ZBDD represents all the solutions at once within a storage space roughly proportional to the number of solutions.

The N-Queens problem is one of the instances of *CSP (Constraint Satisfaction Problems)*. To solve such problems efficiently, Okuno, et al.[26] defined the special operations for ZBDDs, called *restriction*, *exclusion*, and *permission*. Those operations are implemented with the basic operations of ZBDDs and executed efficiently using the cache technique. They improved the perfomance of solving N-Queens problem up to 4.5 times faster than using only the basic operations. They also reported that 4×4 *Magic Square problem* can be solved within 83,852 nodes of ZBDDs, which is 6 times smaller than using the basic operations.

Knight's Tour Problem

Knight's Tour problem is the other famous combinatorial chess problem, which has 400 years long history. The knight's graph contains n^2 vertices representing the squares of the chessboard and the edges describe the legal moves of a knight. A knight's tour is a closed path visiting each square of the chessboard exactly once. M. Löbbing and I. Wegener[13] attacked the problem to compute the number of knight's tours on 8×8 chessboard. They assigned a Boolean variable for each edge in the knight's graph, in total 156 variables to represent all the edges. A solution of knight's tour is represented by a 156-bit combination vector. If we succeed in constructing a ZBDD for the set of all solutions, the number of knight's tour can easily be counted. Unfortunately, the ZBDD is still too large to solve it straightforwardly. They divided the problem into two parts of the chessboad, and then constructed a kind of OBDDs for each sub-space. After minute discussion for connecting the two sub-problems, they succeeded in counting the exact number of 8×8 knight's tour.

In this application, each solution contains 64 edges from the 156 edges. The ratio is about 40%, not very sparse. We can expect that the benefit of ZBDDs is not so remarkable here.

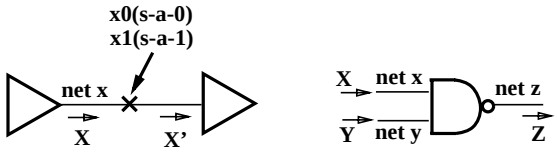
Fault Simulation

Takahashi et al.[29] proposed a fault simulation method considering multiple faults by using OBDDs. it is a deductive method for multiple faults, that manipulates sets of multiple stuck-at faults. It propagates the fault sets from primary inputs to primary outputs, and eventually obtains the detectable faults at primary outputs. Takahashi et al. used ordinary OBDDs, but we can compute the fault simulation more simply by using ZBDDs based on unate cube set algebra.

First, we generate the whole set of multiple faults that is assumed in the simulation. The set F_1 of all the single stuck-at faults is expressed as

$$F_1 = (a_0 + a_1 + b_0 + b_1 + c_0 + c_1 + ...),$$

where a_0 and a_1 represent the stuck-at-0 and -1 faults, respectively, at the net a. Other literals are expressed similarly. We can represent the set F_2 of double and single faults as $(F_1 * F_1)$. Further more, $(F_2 * F_1)$ gives the set of three or fewer multiple faults. If we assume exactly double (not including single) faults, we can calculate $(F_2 - F_1)$. In this way, the whole set U can easily be described with unate cube set expressions.



(a) At the net. (b) At the gate.

Fig. 18. Propagation of fault sets.

After computing the whole set U, we then propagate the detectable fault set from the primary inputs to the primary outputs. As illustrated in Fig. 18(a), two faults x_0 and x_1 are assumed at a net x. Let X and X' be the detectable fault sets at the source and sink, respectively, of the net-x. We can calculate X' from X with the following unate cube expressions:

$$X' = (X + (U/x_1) * x_1) \% x_0, \text{ when net-x is normally 0.}$$
$$X' = (X + (U/x_0) * x_0) \% x_1, \text{ when net-x is normally 1.}$$

The first expression means that: if net-x should be 0 in the normal circuit, the multiple faults with x_1 (stack-at-1 at x) are detectable but the multiple faults with x_0 (stack-at-0 at x) cannot be detected hereafter. The second expression shows the other case similarly.

On each gate, we calculate the fault set at the output of the gate from the fault sets at the inputs of the gate. Let us consider a NAND gate with two inputs x and y, and one output z, as shown in Fig. 18(b). Let X, Y, and Z be the fault sets at x, y, and z. We can calculate Z from X and Y by the simple unate cube set operations as follows:

$$Z = X \cap Y, \text{ when } x = 0, y = 0, z = 1 \text{ in normal circuit.}$$
$$Z = X - Y, \text{ when } x = 0, y = 1, z = 1 \text{ in normal circuit.}$$
$$Z = X + Y, \text{ when } x = 1, y = 1, z = 0 \text{ in normal circuit.}$$

The first expression means that: if $x = 0, y = 0$, and $z = 1$ in the normal circuit, both x and y must be inverted to make $z = 0$. Therefore, only the faults commonly included in X and Y can survive in Z. The second and third expressions shows the other cases.

In this way, we can compute the detectable fault sets by calculating those expressions for all the gates in the

circuit. Using unate cube set algebra, we can simply describe the fault simulation procedure and can execute it directly by using ZBDD operations.

Unate cube set expressions are suitable for representing sets of combinations, and they can be efficiently manipulated using ZBDDs. For solving some types of combinatorial problems, ZBDDs are more efficient than those using original OBDDs. We expect the unate cube set calculation is useful in developing VLSI CAD systems and in various other applications.

4 Multi-Level Logic Synthesis Using ZBDDs

Logic synthesis and optimization techniques have been used successfully for practical design of VLSI circuits in recent years. Multi-level logic, Optimization is important in logic synthesis systems and a lot of research in this field has been undertaken[23,14]. In particular, the *algebraic logic minimization method*, such as MIS[2], is the most successful and prevalent way to attain this optimization. It is based on cube set (or two-level logic) minimization and generates multi-level logic from cube sets by applying a *weak-division method*. This approach is efficient for functions that can be expressed in a feasible size of cube sets, but we are sometimes faced with functions whose cube set representations grow exponentially with the number of inputs. Parity functions and full-adders are examples of such functions. This is a problem of the cube-based logic synthesis methods.

As noted in previous sections, the OBDD-based technique[4] provided a break-through for that problem. The implicit cube set manipulation method greatly reduces the computation time and memory requirement. For such applications, ZBDDs are especially suitable.

In this section, we discuss an application of ZBDDs for multi-level logic synthesis. We first define the operations of *binate cube sets*, and show a fast method[15] for generating irredundant cube sets for given Boolean functions. We also describe the fast factorization method[19] of the cube sets. These algorithm can be computed in a time almost proportional to the number of nodes in ZBDDs, which are usually much smaller than the number of literals in the cube set. In this method, we can quickly generate multi-level logic from cube sets even for parity functions and full-adders, that have not been possible to handle when using the conventional algebraic methods.

4.1 Implicit Cube Set Representation

Cube sets (also called covers, PLAs, sum-of-products forms, and two-level logic) are used to represent Boolean functions in many problems in the design and testing of digital systems. In a cube set, each cube is formed by a combination of positive and negative literals for input variables. (We are speaking here of a *binate* cube set, different from the *unate* cube set discussed in Section 3.)

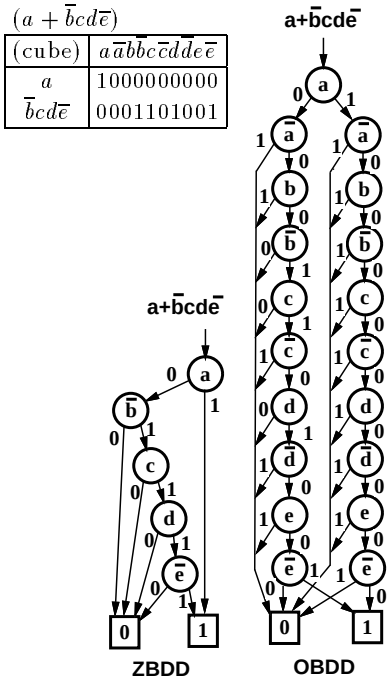


Fig. 19. ZBDD and OBDD for representing a cube set.

Figure 19 illustrates ZBDD-based representation of a cube set of positive and negative literals. In this representation, we use twice number of input variables for the two kind of literals. var and $\overline{var}$ never appear together in the same cube, and at least one should be 0. The 0's are automatically suppressed by using ZBDDs. This is the main advantage of using ZBDDs comparing to ordinary OBDDs.

The basic operations for the binate cube set representation are the following:

"$\emptyset$"	the empty set. (0-terminal node.)
"**1**"	the set of only the tautology cube. (1-terminal node.)
P.top	the input variable of the literal at the root node.
P.and0(v)	appends literal $\overline{v}$ for each cube.
P.and1(v)	appends the literal v for each cube.
P.factor0(v)	selects the cubes including $\overline{v}$, and then delete $\overline{v}$ from each combination.
P.factor1(v)	selects the cubes including v, and then delete v from each combination.
P.factorD(v)	selects the cubes not including $v, \overline{v}$.
$P + Q$	the union set of P and Q.
$P \cap Q$	the intersection set of P and Q.
$P - Q$	the difference set. (cubes in P but not in Q.)
P.cubes	the number of cubes in P.
P.literals	the number of total literals in P.

GetISOP($f(X)$): /* $f(X) : \{0,1\}^n \rightarrow \{0,1,d\}$*/
  if ($f(X)$ always returns 0 or d) $isop \leftarrow 0$
  else if ($f(X)$ always returns 1 or d) $isop \leftarrow 1$
  else {
    $v \leftarrow$ one of variable in X
      /* choose top variable in BDD. */
    $f_1(X') \leftarrow f(X|_{v=1})$ /* $X' = X - \{v\}$. */
    $f_0(X') \leftarrow f(X|_{v=0})$ /* f_1, f_0 are cofactors of f. */
    $f_P(X') \leftarrow \begin{cases} d & \text{if } (f_1(X')=1) \wedge (f_0(X') \neq 0) \\ f_1(X') & \text{otherwise} \end{cases}$
      /* f_P must be covered by cubes with v. */
    $isop_P \leftarrow$ GetISOP($f_P(X')$)
      /* generates $isop_P$ recursively. */
    $f_N(X') \leftarrow \begin{cases} d & \text{if } (f_0(X')=1) \wedge (f_1(X') \neq 0) \\ f_0(X') & \text{otherwise} \end{cases}$
      /* f_N must be covered by cubes with $\overline{v}$. */
    $isop_N \leftarrow$ GetISOP($f_N(X')$)
      /* generates $isop_N$ recursively. */
    $f_1'(X') \leftarrow \begin{cases} d & \text{if already covered by } isop_P \\ f_1(X') & \text{otherwise} \end{cases}$
    $f_0'(X') \leftarrow \begin{cases} d & \text{if already covered by } isop_N \\ f_0(X') & \text{otherwise} \end{cases}$
    $f_D(X') \leftarrow \begin{cases} 0 \text{ if } (f_1'(X')=0) \vee (f_0'(X')=0) \\ d \text{ if } (f_1'(X')=d) \wedge (f_0'(X')=d) \\ 1 \text{ otherwise} \end{cases}$
      /* f_D must be covered by cubes without $v, \overline{v}$. */
    $isop_D \leftarrow$ GetISOP($f_D(X')$)
      /* generates $isop_D$ recursively. */
    $isop \leftarrow v \cdot isop_P + \overline{v} \cdot isop_N + isop_D$
  }
  return $isop$

Fig. 20. Algorithm for generating prime-irredundant cube sets

Any cube can be generated by applying a number of "and0" or "and1" to the 1-terminal node. Notice that the intersection is different from the logical AND operation; it extracts only the common cubes in the two cube sets. These operations are simply composed of ZBDD operations, and their execution time is roughly proportional to the size of the graphs.

4.2 Generation of Prime-Irredundant Cube Sets

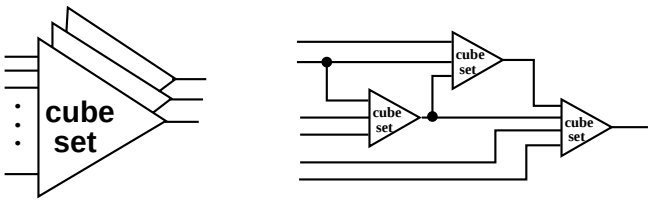
Cube set representation does not give unique forms of Boolean functions, and it is very important issue to find the minimal or nearly minimal form for a given Boolean function.

Minato[15] presented a fast algorithm to generate a prime-irredundant[3] cube set directly from a given OBDD. This algorithm is based on *recursive operator* shown by Morreale[21], and we call it *Minato-Morreale algorithm*. The procedure is described in Fig. 20.

This algorithm is summarized in the expansion:

$$isop = \overline{v} \cdot isop_0 + v \cdot isop_1 + isop_d$$

[3] Any one literal or cube cannot be deleted to keep the function. Not always the minimum set but nearly minimum in most cases.



(a) Initial cube sets. (b) Final cube set network.

Fig. 21. Factorization of cube sets.

where $isop$ represents the prime-irredundant cube set, and v is one of the input variables. This expansion reveals that $isop$ can be divided into three subsets containing $\overline{v}$, v, and the others. When $\overline{v}$ and v are excluded from each cube, the three subsets of $isop_1$, $isop_0$ and $isop_d$ should also be prime-irredundant. Based on this expansion, the algorithm generates a prime-irredundant cube set recursively.

At first the program is implemented with linear list representation of cube sets, but later it is remarkably accelerated by using ZBDDs. The key technique is the hash-based cache to store the results of each recursive call. By checking this cache, we can avoid duplicate execution for the shared subgraphs in the ZBDD. The experimental result in the paper[19] demonstrates that the practical benchmark circuits are flattened into cube sets with billions of literals in only several minutes. Those Boolean functions have never been flattened by using previous data structures. Since cube set is a basic mathematical model, ZBDD-based simplification method will be useful not only for VLSI CAD but also for many other problems in computer science.

4.3 Factorization of Cube Sets

In general, two-level logics can be factorized into more compact multi-level logics. The initial two-level logics are represented with large cube sets for primary output functions, as shown in Fig. 21(a). When we determine a good intermediate logic, we make a cube set for it and reduce the other existing cube sets by using a new intermediate variable. Eventually, we construct a multi-level logic that consists of a number of small cube sets, as illustrated in Fig 21(b). The multi-level logic consists of hundreds of cube sets, each of which is very small. On the average, less than 10 variables out of hundreds are used for each cube set. They yield so sparse combinations that the use of ZBDDs is quite effective. Another benefit of ZBDDs is that we do not have to fix the number of variables beforehand. We can use additional variables whenever an intermediate logic is found.

Weak-division (or algebraic division) is the most successful and prevalent method for generating multi-level logics from cube sets. For example, the cube set expres-

```
procedure(F/P)
{   if (P = 1) return F ;
    if (F = 0 or F = 1) return 0 ;
    if (F = P) return 1 ;
    Q ← cache("F/P") ; if (Q exists) return Q ;
    v ← P.top ; /* the highest variable in P */
    (F0, F1, Fd) ← factors of F by v ;
    (P0, P1, Pd) ← factors of P by v ;
    Q ← P ;
    if (P0 ≠ 0) Q ← F0/P0 ;
    if (Q = 0) return 0 ;
    if (P1 ≠ 0)
        if (Q = P) Q ← F1/P1 ;
        else Q ← Q ∩ (F1/P1) ;
    if (Q = 0) return 0 ;
    if (Pd ≠ 0)
        if (Q = P) Q ← Fd/Pd ;
        else Q ← Q ∩ (Fd/Pd) ;
    cache("F/P") ← Q ;
    return Q ;
}
```

Fig. 22. Implicit weak-division algorithm.

```
procedure(F · G)
{   if (F.top < G.top) return (G · F) ;
    if (G = 0) return 0 ;
    if (G = 1) return F ;
    H ← cache("F · G") ; if (H exists) return H ;
    v ← F.top ; /* the highest variable in F */
    (F0, F1, Fd) ← factors of F by v ;
    (G0, G1, Gd) ← factors of G by v ;
    H ← v̄(F0 · G0 + F0 · Gd + Fd · G0)
      +v(F1 · G1 + F1 · Gd + Fd · G1) + Fd · Gd ;
    cache("F · G") ← H ;
    return H ;
}
```

Fig. 23. Implicit multiplication algorithm.

sion

$$F = a\ b\ d + a\ b\ \overline{e} + a\ b\ \overline{g} + c\ d + c\ \overline{e} + c\ h$$

can be divided by $(a\ b + c)$. By using an intermediate variable p, we can rewrite the expression:

$$F = p\ d + p\ \overline{e} + a\ b\ \overline{g} + c\ h \quad p = a\ b + c.$$

In the next step, f will be divided by $(d + \overline{e})$ in a similar manner.

Weak-division does not exploit all of Boolean properties of the expression and is only an algebraic method. In terms of result quality, it is not as effective as other stronger optimizing methods, such as the *transduction method*[23]. However, weak-division is still important because it is used for generating initial logic circuits for other strong optimizers, and is applied to large-scale logics that cannot be handled by strong optimizers.

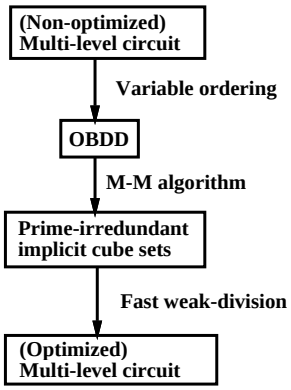


Fig. 24. Basic flow of multi-level logic synthesizer.

As described in previous sections, the weak-division operation can efficiently be executed by using ZBDDs. The algorithm for unate cube sets is already shown in Section 3. Here we explain the division algorithm for binate cube sets. The procedure is described in Fig. 22. The basic idea is summarized as follows.

For the variable $v = P$.top (not use F.top), the cube sets F and P are factored into three parts as:

$$F = \overline{v}\ F_0 + v\ F_1 + F_d$$
$$P = \overline{v}\ P_0 + v\ P_1 + P_d.$$

The quotient (f/p) can then be written as

$$(F/P) = (F_0/P_0) \cap (F_1/P_1) \cap (F_d/P_d).$$

Each sub-quotient term can be computed recursively. The procedure is eventually broken down into trivial problems and the results are obtained. If one of the values for P_0, P_1, or P_d is zero, we may skip the term. For example, if $P_1 = 0$, then $(F/P) = (F_0/P_0) \cap (F_d/P_d)$. Whenever we find that one of the values for (F_0/P_0), (F_1/P_1) and (F_d/P_d) becomes zero, $(F/P) = 0$ becomes obvious and we no longer need to continue the calculation.

In the same way as for the Minato-Morreale algorithm, we prepared a hash-based cache to store results for each recursive call and avoid duplicate execution. Using the cache technique, we can execute this algorithm in a time nearly proportional to the size of the graph, regardless of the number of cubes and literals.

To obtain the remainder of division $(F\%P) = F - P\ (F/P)$, we need to compute the algebraic multiplication between two cube sets. This procedure can also be described recursively and executed quickly using the cache technique, as illustrated in Fig. 23.

4.4 Implementation and Evaluation

Based on the above methods, Minato[19] implemented a multi-level logic synthesizer illustrated in Fig. 24. Starting with non-optimized multi-level logics, the OBDDs

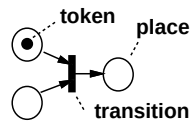
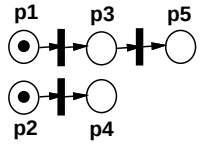


Fig. 25. A Petri net.



(a) Initial marking.

p_1	p_2	p_3	p_4	p_5
1	1	0	0	0
1	0	0	1	0
0	1	1	0	0
0	1	0	0	1
0	0	1	1	0
0	0	0	1	1

(b) Reachable marking set.

Fig. 26. Reachable marking set for a Petri net.

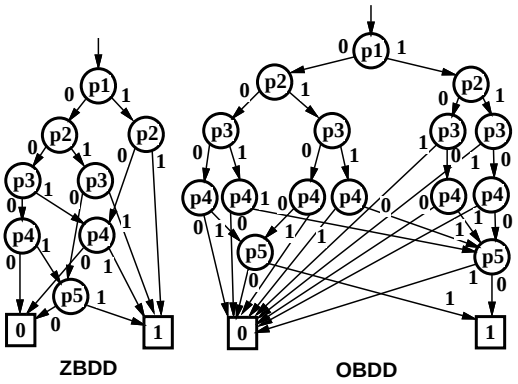


Fig. 27. ZBDD and OBDD for the reachable marking set.

for the Boolean functions of primary outputs are constructed under a heuristic ordering of input variables. Next the OBDDs are transformed into prime-irredundant implicit cube sets. The cube sets are then factorized into optimized multi-level logics by using the fast weak-division method. The experimental results in the paper[19] show that we can quickly flatten and factorize circuits, even for large-scale parity functions and adders, that have never been flattened with other methods. The program runs much faster than using previous methods, and the difference is remarkable (10 or 100 times faster) when large cube sets are factorized. This result demonstrates the effect of the implicit cube set representation. ZBDD-based method greatly accelerates the logic synthesis process and enlarges the scale of the circuits to which these systems are applicable.

There is still room to improve the results. The strategy of choosing divisors is another hard problem as well as performing division. In addition, more strong division method for implicit cube sets is worth investigating to improve the optimization quality.

5 Other ZBDD Applications

5.1 Processing of Petri nets

Petri nets[22] have been used to model various concurrent systems such as network protocols, asynchronous circuits, finite state machines, etc. As shown in Fig. 25, a Petri net consists of a finite set of *places*, a finite set of transitions representing the flow relation, and a number of *tokens* representing the current state. A pattern of tokens is called a *marking*.

Yoneda, et al.[31] proposed a method of manipulating the marking set of a Petri net using ZBDDs. Figure 26(a) shows the initial marking of an example of Petri net. There are some choices of transitions, and thus a number of markings are possible from this initial state. The set of all reachable markings[4] are enumerated in Fig.26(b). This is a kind of set of combinations, and it can be represented by ZBDDs. Figure 27 shows the ZBDD and OBDD representation for the same reachable marking set. Since the marking sets often include sparse combinations of tokens, ZBDDs are useful in this applications. In the paper they present the algorithms of computing the reachable marking sets using ZBDDs. Their experimental results show that the reachable set containing 900 million of markings can be represented with 500k nodes of ZBDDs. The size ZBDDs are 2 or 3 times smaller than using original OBDDs, and the computation time can be reduced up to 30 times less.

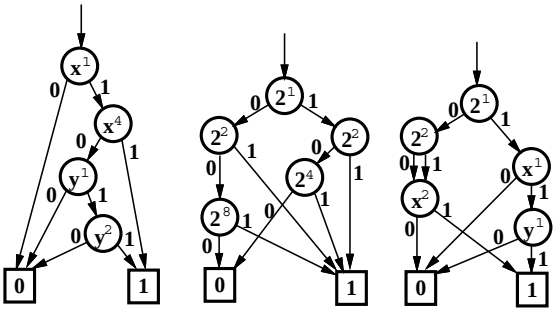
5.2 Framework for Graph Optimization

O. Coudert[7] presented a framework for general graph optimization problems based on ZBDD manipulation. The framework can be used for the problems as follows.

- Maximal cliques
- Maximum k-cover
- Minimum α-covering
- Maximum independent set
- Minimum vertex cover
- Minimum coloring
- Minimum clique partition
- Minimum clique cover
- Minimum dominant set
- Minimum edge dominating set

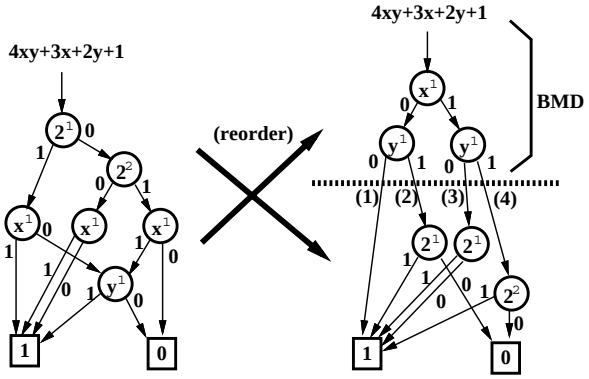
In the paper, he defined a special operations of ZBDDs, called *NotSupSet, NotSubSet, AllEdge, MaxSet*, and *MaxDot*, to improve the performance of solving these problems. These operations can be implemented with the basic operations of ZBDDs and executed efficiently using the cache technique. He also presented an example of application to the routing problem in LSI CAD. This framework will be useful for many combinatorial problems in various areas.

[4] Here we consider only *one-safe* Petri nets, where at most one token can exist at a place.



(a) $x^5 + x^1y^3$ (b) $2^8 + 2^5 + 2^3 + 2^2$ (c) $5x^2 + 2xy$

Fig. 28. ZBDD-based polynomial representation.



(a) Coefficients are higher. (b) Coefficients are lower.

Fig. 29. Comparison of ZBDD-based method and BMD.

5.3 Manipulation of Polynomial Formulas

A polynomial formula is a basic model in mathematics. Minato[18] presented a method for representing polynomials using ZBDDs. It is similar to the cube set representation, except that they include higher degree variables and numerical coefficients.

The basic idea is based on binary coding of higher degrees and coefficients. A variable x^k can be broken down into:

$$x^k = x^{(k_1+k_2+\ldots+k_m)} = x^{k_1}x^{k_2}\ldots x^{k_m},$$

where $k_1, k_2, \ldots, k_m$ are different powers of 2. In this way, we can represent x^k as a combination of $x^1, x^2, x^4, x^8, \cdots, x^{2^{n-1}} (0 < k < 2^n)$. Such combinations can be dealt with efficiently using ZBDDs. For example, a polynomial $x^5 + xy^3$ can be written as $x^1x^4 + x^1y^1y^2$. It can be regarded as a cube set over x^1, x^4, y^1, and y^2. The formula, then, can be represented by using a ZBDD, as shown in Fig. 28(a).

Numerical coefficients can also be represented similarly. An integer c (> 1) can be written as a sum of powers of 2:

$$c = 2^{c_1} + 2^{c_2} + \ldots + 2^{c_m},$$

where $c_1, c_2, \ldots, c_m$ are different nonnegative integers. Then, regarding "2" as a variable, just like x, y, z, etc., it can be represented as a polynomial of variables with degrees. Consequently, we can represent a constant number c as a cube set over $2^1, 2^2, 2^4, 2^8, \ldots, 2^{2^{n-1}} (0 < c < 2^{2^n})$ using ZBDDs. For example, the constant number $300 = 2^8+2^5+2^3+2^2$ can be written as $2^8+2^12^4+2^12^2+2^2$, and represented by a ZBDD as Fig. 28(b). When the number is used for a coefficient with other variables, we can regard the symbol "2" just as one sort of variable in the formula. Figure 28(c) shows an example for $5x^2 + 2xy$, which is decomposed into $x^2 + 2^2x^2 + 2^1x^1y^1$.

In this method, we can compactly and uniquely represent large-scale polynomials with millions of terms, and can manipulate them in practical time. Constructing canonical forms of polynomials immediately leads to equivalence checking of arithmetic expressions.

BMDs[3], also discussed in the overview paper[9], can also represent polynomials. One big difference is that BMDs assume binary-valued variables, so BMDs cannot deal with the higher-degree variables. Except for this difference, the two representations are similar to each other. Fig. 29(a) shows the ZBDD-based representation for $(4xy + 3x + 2y + 1)$. If we change the variable order such that the coefficient variables move from higher to lower position, the ZBDD becomes as shown in Fig. 29(b). In this graph, the sub-graphs with the coefficient variables correspond to the terminal nodes in the BMD. This observation indicates that the ZBDD-based representation and the BMD can be transformed into each other by changing the variable order.

6 Conclusion

In this article, we discussed the basic data structures and algorithms of ZBDDs, and surveyed their applications. The ZBDD programs are now widely implemented in many publicly available BDD packages, such as the CUDD[28], CAL-2.0[27], and the TiGeR[6] library.

Here we summarize the four check points for using ZBDDs effectively.

- Dealing with many input variables.
  (Reduction factor is bounded by the number of inputs.)
- Variables are regarded as zero in default.
- Representing sets of "sparse" combinations.
  (This is different from "sparse" sets of combinations.)
- Containing a large number of combinations.
  (Otherwise, linear-list representation is enough to use.)

ZBDDs are suitable to the applications which satisfy the above conditions. We should determine the property of data before choosing a type of OBDDs. Even for the case where ZBDDs are suitable, just replacing the data structure is not enough to have the benefit of ZBDDs. We should consider a better sequence of operations for ZBDDs.

When we use ZBDDs considering the above properties, the performance of programs can be improved in terms of time and space, and programming will be easier than using ordinary OBDDs. We conclude that ZBDD is a useful option in the OBDD techniques, suitable for some kinds of practical applications.

References

1. B. Becker, R. Drechsler and M. Theobald. On the Implementation of a Package for Efficient Representation and Manipulation of Functional Decision Diagrams. *IFIP WG 10.5 Workshop on Applications of the Reed-Muller Expression in Circuit Design*, pp. 162–169, 1993.
2. R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli and A. R. Wang. MIS: A Multiple-Level Logic Optimization System. *IEEE Trans. on Computer-Aided Design*, Vol. CAD-6, No. 6, pp. 1062–1081, November 1987.
3. R. E. Bryant and Y.-A. Chen. Verification of Arithmetic Functions with Binary Moment Diagrams. *Proc. 32nd ACM/IEEE Design Automation Conf. (DAC-95)*, pp. 535–541, June 1995.
4. O. Coudert and J. C. Madre. Implicit and Incremental Computation of Primes and Essential Primes of Boolean Functions. In *Proc. of 29th ACM/IEEE Design Automation Conf. (DAC-92)*, pp. 36–39, June 1992.
5. O. Coudert, J. C. Madre, and H. Fraisse. A New Viewpoint of Two-Level Logic Optimization. In *Proc. of 30th ACM/IEEE Design Automation Conf. (DAC93)*, pp. 625–630, June 1993.
6. O. Coudert, J. C. Madre and H. Touati. *TiGeR Version 1.0 User Guide*. Digital Paris Research Lab., December 1993.
7. O. Coudert. Solving Graph Optimization Problems with ZBDDs. *Proc. of ACM/IEEE Europian Design and Test Conf. (ED&TC-97)*, 1997.
8. R. Drechsler, M. Theobald and B. Becker. Fast OFDD Based Minimization of Fixed Polarity Reed-Muller Expressions. *Proc. of European Conf. on Design Automation*, pp. 2–7, 1994.
9. R. Drechsler and D. Sieling. Binary Decision Diagrams in Theory and Practice. *Software Tools for Technology Transfer*, This special section, 1999.
10. U. Kebschull, E. Schubert and W. Rosenstiel. Multilevel Logic Synthesis Based on Functional Decision Diagrams. *Proc. of European Conf. on Design Automation*, pp. 43–47, March 1992.
11. H.-T. Liaw and C.-S. Lin. On the OBDD-Representation of General Boolean Functions. *IEEE Trans. on Computers*, Vol. C-41, No. 6, pp. 661–664, June 1992.
12. B. Lin and F. Somenzi. Minimization of Symbolic Relations. In *Proc. of IEEE/ACM International Conference on Computer-Aided Design (ICCAD-90)*, pp. 88–91, November 1990.
13. M. Löbbing and I. Wegener. The Number of Knight's Tour Equals 33,439,123,484,294 — Counting with Binary Decision Diagrams. *The Electoric Journal of Combinatorics*, Vol. 3:#R5, 1996.
14. Y. Matsunaga and M. Fujita. Multi-Level Logic Optimization Using Binary Decision Diagrams. *Proc. of IEEE/ACM International Conference on Computer-Aided Design (ICCAD-89)*, pp. 556–559, November 1989.
15. S. Minato. Fast Generation of Irredundant Sum-of-Products Forms from Binary Decision Diagrams. *Proc. of Synthesis and Simulation Meeting and International Interchange (SASIMI-92, Kobe, Japan)*, pp. 64–73, April 1992.
16. S. Minato. Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems. In *Proc. of 30th ACM/IEEE Design Automation Conference (DAC-93)*, pp. 272–277, June 1993.
17. S. Minato. Calculation of Unate Cube Set Algebra Using Zero-Suppressed BDDs. *Proc. of 31st ACM/IEEE Design Automation Conference (DAC-94)*, pp. 420–424, June 1994.
18. S. Minato. Implicit Manipulation of Polynomials Using Zero-Suppressed BDDs. *Proc. of ACM/IEEE Europian Design and Test Conf. (ED&TC-95)*, pp. 449-454, Mar. 1995.
19. S. Minato. Fast Factorization Method for Implicit Cube Set Representation. *IEEE Trans. on CAD*, 15(4), pp. 377–384, Apr. 1996.
20. S. Minato. Arithmetic Boolean Expression Manipulator Using BDDs. *Formal Methods in System Design*, Kluwer Academic Publishers, Vol. 10, pp. 221–242, 1997.
21. E. Morreale. Recursive Operators for Prime Implicant and Irredundant Normal Form Determination. *IEEE Trans. Comput.*, C-19(6), pp. 504–509, June 1970.
22. T. Murata. Petri Nets: Properties, Analysis and Applications. *Proc. of the IEEE*, Vol. 77(4), pp. 541-580, 1989.
23. S. Muroga, Y. Kambayashi, H. C. Lai and J. N. Culliney. The Transduction Method—Design of Logic Networks Based on Permissible Functions. *IEEE Trans. on Computers*, Vol. C-38, No. 10, pp. 1404–1424, October 1987.
24. H. Ochi, K. Yasuoka and S. Yajima. Breadth-First Manipulation of Very Large Binary-Decision Diagrams. In *Proc. of IEEE/ACM International Conference on Computer-Aided Design (ICCAD-93)*, pp. 48–55, November 1993.
25. H. G. Okuno. Reducing Combinatorial Explosions in Solving Search-Type Combinatorial Problems with Binary Decision Diagram. *Trans. of Information Processing Society of Japan (IPSJ), (in Japanese)*, Vol. 35, No. 5, pp. 739–753, May 1994.
26. H. G. Okuno, S. Minato and H. Isozaki. On the Properties of Combination Set Operations. *Information Processing Letters*, Vol.66, pp. 195–199, 1998.
27. R. K. Ranjan and J. Sanghavi. CAL-2.0: Breadth-First Manipulation Based BDD Library. *Public software*. Univ. of California, Berkeley, CA, USA, June 1997. `http://www-cad.eecs.berkeley.edu/Research/cal_bdd/`
28. F. Somenzi. CUDD: CU Decision Diagram Package. *Public Software*. University of Colorado, Boulder, CO, USA, April 1997. `http://vlsi.colorado.edu/~fabio/CUDD`
29. N. Takahashi, N. Ishiura and S. Yajima. Fault Simulation for Multiple Faults Using BDD Representation of Fault Sets. *Proc. of IEEE/ACM International Conference on Computer-Aided Design (ICCAD-91)*, pp. 550–553, November 1991.
30. I. Wegener. The Size of Reduced OBDD's and Optimal Read-Once Branching Programs for Almost All Boolean Functions. *IEEE Trans. on Computers*, Vol. C-43, No. 11, pp. 1262–1269, November 1994.
31. T. Yoneda, H. Hatori, A. Takahara and S. Minato. BDDs vs. Zero-Suppressed BDDs: for CTL Symbolic Model

Checking of Petri Nets. *Proc. of 1st International Conference on Formal Method in Computer Design (FMCAD-96, Palo Alto, CA, USA*, pp. 435–449, November, 1996.

Appendix: Proof of Theorem 1

For a given (ordinary) OBDD, we reversely apply the node deletion (S-deletion) rule to re-insert all redundant nodes. After that, only the merging rule is effective in the OBDD. This type of OBDD is called *quasi-reduced OBDD*[24,30]. It is known that the quasi-reduced OBDDs are still canonical for Boolean functions under a fixed input domain and variable ordering. We then apply the zero-suppressed (pD-deletion) rule to the quasi-reduced OBDD, and obtain the completely reduced ZBDD. In this process, we never create two identical nodes in the same variable level because the merging rule is always effective. This implies that a ZBDD uniquely represents a Boolean function if the input domain and variable ordering are fixed. (Q.E.D.)