Calculation of Unate Cube Set Algebra Using Zero-Suppressed BDDs

Shin-ichi Minato
NTT LSI Laboratories
3-1 Morinosato-W akamiy a, Atsugi-shi, Kanagaw a Pref., 243-01 Japan

Abstract — Many com binatorial problems in LSI design can be described with cube set expressions. W e discuss unate cube set algebra based on zero-suppressed BDDs, a new type of BDDs adapted for cube set manipulation. W e propose efficient algorithms for computing unate cube set operations including multiplication and division, followed by some practical applications of unate cube set calculation.

1 Introduction

Recently, Binary Decision Diagrams (BDDs), which are graph-based representations of Boolean functions[1], have attracted much attention because they enable us to manipulate Boolean functions efficiently in terms of time and space. There are many cases that the algorithm based on conventional data structures can be significantly improved by using BDDs[2][3].

As our understanding of BDDs has deepened, the range of applications has broadened. Besides Boolean functions, we are often faced with manipulating *sets of combinations* in VLSI CAD problems. By mapping a set of combinations into the Boolean space, they can be represented as a characteristic function using a BDD. This method enables us to implicitly manipulate a huge number of combinations, which have never been practical before. Recently, new two-level logic minimization methods have been developed based on implicit set representation[4]. These techniques are also used to solve general covering problems[5].

BDD-based set representation is more efficient than conventional methods. However, it can be inefficient at times because BDDs were originally designed to represent Boolean functions. W e have recently developed a new type of BDD which has been adapted for set representation[6]. This type of BDD, called a *zero-suppressed BDD* (0-sup-BDD), enables us to represent sets of com binations more efficiently than using conventional ones. It is useful for solving combinatorial problems that sometimes arise in VLSI design.

When describing algorithms or procedures for manipulating BDDs, we usually use Boolean expressions based on switching algebra. Similarly, when considering sets of combinations with 0-sup-BDDs, we can use *unate cube set* expressions and their algebra. Based on unate cube set algebra, we can simply describe algorithms or procedures for 0-sup-BDDs. W e developed efficient algorithms for executing unate cube set operations including multiplication and division.

In this paper, we discuss calculation of unate cube set algebra using 0-sup-BDDs. W e propose efficient algorithms for computing unate cube set operations and some practical applications.

2 Zero-Suppressed BDDs

Zero-suppressed BDDs (0-Sup-BDDs)[6] are a new type of BDDs[6] that are adapted for representing sets of combinations. They are based on the following reduction rules:

- Eliminate all nodes with 1-edge pointing to the 0-terminal node. Then connect the edge to the other subgraph directly (Fig. 1).
- Share all equivalent sub-graphs in the same manner as that for conventional BDDs.

Notice that, contrary to conventional BDDs, we do not eliminate nodes whose two edges point to the same node. This reduction rule is asymmetric for the two edges because the nodes remain when their 0-edge points to a terminal node. When the num ber and order of the variables are fixed, 0-sup-BDDs provide canonical forms for Boolean functions.

Fig. 2 illustrates conventional and 0-Sup-BDDs representing the sets of combinations. Using the "0-sup" reduction rule, the two BDDs are automatically reduced into the same form, free of irrelevant variables. 0-sup-BDDs are more effective for sparser combinations, which means only a few objects out of many are included in each combination in the set.

The methods for manipulating 0-sup-BDDs are defined as set operations and differ slightly from conventional

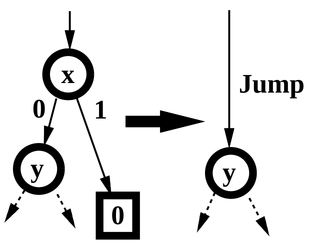


Fig. 1: Reduction Rule for 0-sup-BDDs

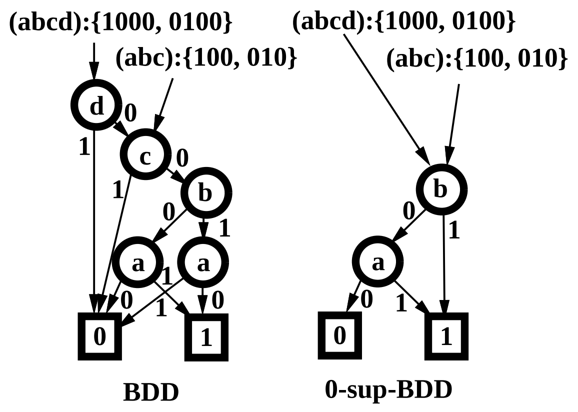


Fig. 2: Effect of "0-sup" Reduction Rule

BDD manipulation. First, we generate trivial graphs and then construct more complex ones by applying basic operations such as union, intersection, and difference. We can execute these operations in a time almost proportional to the size of the graphs, just as using conventional BDDs. The basic operations for 0-sup-BDDs are summarized below: (see [6] for detailed algorithms.)

Empty()	returns ϕ. (empty set)
Base()	returns $\{\mathbf{0}\}$.
Subset1(P, var)	returns the subset of P such as $var = 1$.
Subset0(P, var)	returns the subset of P such as $var = 0$.
Change(P, var)	returns P when var is inverted.
Union(P, Q)	returns $(P \cup Q)$
Intsec(P, Q)	returns $(P \cap Q)$
Diff(P, Q)	returns $(P - Q)$
Count(P)	returns $\lvert P\rvert$. (number of elements)

Using the above operations, we can deal with any set of combinations using 0-sup-BDDs; however, such C-like notation is not so easy to use. A simpler method is desired.

3 Unate Cube Set Algebra

In this section, we discuss unate cube set algebra for manipulating sets of combinations. 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. Each cube represents one combination, and each literal represents an object chosen in the combinations.

We sometimes use cube sets to represent Boolean functions; however, they are usually *binate* cube sets containing negative literals. Binate cube sets have different semantics from unate cube sets. In binate cube sets, literal x and $\overline{x}$ represent $x = 1$ and $x = 0$, respectively, while the absence of a literal means *don't care*, namely $x = 1, 0$, both OK. On the other hand, in unate cube sets, literal x means $x = 1$ and an absence means $x = 0$. For example, the cube set expression $(a + bc)$ represents $(abc) : \{111, 110, 101, 100, 011\}$ under the semantics of binate cube sets, but $(abc) : \{100, 011\}$ under unate cube set semantics.

3.1 Basic Operations

Unate cube set expressions consist of trivial sets and algebraic operators. There are three kinds of trivial sets:

0	(empty set),
1	(unit set),
x_k	(single literal set).

The unit set "1" includes only one combination that chooses no objects. This set becomes the unit element of the product operation. A single literal set x_k includes only one combination that chooses only one literal. In the following section, a lower-case letter denotes a literal, and an upper-case letter denotes a cube set expression.

We arranged the line-up of the basic operators as follows:

$\&$	(intersection),
$+$	(union),
$-$	(difference),
$*$	(product),
$/$	(quotient of division),
$\%$	(remainder of division).

(Sometimes we omit "$*$". We may use a comma "," instead of "+".) For example, $\{ab, b, c\} \& \{ab, 1\} = \{ab\}$, $\{ab, b, c\} + \{ab, 1\} = \{ab, b, c, 1\}$ and $\{ab, b, c\} - \{ab, 1\} = \{b, c\}$. It stands that $(P - Q) = (Q - P)$ if and only if $P = Q$.

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

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

In this operation, $P * 0 = 0$, $P * 1 = P$, $P * (Q + R) = (P * Q) + (P * R)$ and $a * a = a$. In general, $P * P \neq P$.

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*[7].

1. When Q includes only one cube, (P/Q) is obtained by extracting a subset of P, which consists of cubes including Q's cube, and then eliminating Q's literals from the subset. 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,
$$\begin{aligned}&\{abd, abe, abg, cd, ce, ch\}/\{ab, c\}\\ &\quad = (\{abd, abe, abg, cd, ce, ch\}/\{ab\})\\ &\qquad \& \ (\{abd, abe, abg, cd, ce, ch\}/\{c\})\\ &\quad = \{d, e, g\} \& \{d, e, h\}\\ &\quad = \{d, e\}\ .\end{aligned}$$
3. $(P\%Q)$ can be obtained by calculating $P - P * (P/Q)$.

These three trivial sets and six basic operators are used to represent and manipulate sets of combinations. In Section 2, we defined other three basic operations of Subset1(), Subset0() and Change() for assigning a value to a literal; however, we do not have to use the three operations since the weak-division operation can be used as *generalized cofactor* for 0-sup-BDDs. For example, Subset1(P, x_k) can be described as $(P/x_k) * x_k$, and

```
procedure($P * Q$)
{   if ($P.top < Q.top$) return ($Q * P$) ;
    if ($Q = 0$) return 0 ;
    if ($Q = 1$) return $P$ ;
    $R \leftarrow$ cache("$P * Q$") ;  if ($R$ exists) return $R$ ;
    $x \leftarrow P.top$ ; /* the highest variable in $P$ */
    $(P_0, P_1) \leftarrow$ factors of $P$ by $x$ ;
    $(Q_0, Q_1) \leftarrow$ factors of $Q$ by $x$ ;
    $R \leftarrow x\ (P_1 * Q_1 + P_1 * Q_0 + P_0 * Q_1) + P_0 * Q_0$ ;
    cache("$P * Q$") $\leftarrow R$ ;
    return $R$ ;
}
```

Fig. 3: Algorithm for Product

```
procedure($P/Q$)
{   if ($Q = 1$) return $P$ ;
    if ($P = 0$ or $P = 1$) return 0 ;
    if ($P = Q$) return 1 ;
    $R \leftarrow$ cache("$P/Q$") ;  if ($R$ exists) return $R$ ;
    $x \leftarrow Q.top$ ; /* the highest variable in $Q$ */
    $(P_0, P_1) \leftarrow$ factors of $P$ by $x$ ;
    $(Q_0, Q_1) \leftarrow$ factors of $Q$ by $x$ ; /* ($Q_1 \neq 0$) */
    $R \leftarrow P_1/Q_1$ ;
    if ($R \neq 0$) if ($Q_0 \neq 0$) $R \leftarrow R\ \&\ P_0/Q_0$ ;
    cache("$P/Q$") $\leftarrow R$ ;
    return $R$ ;
}
```

Fig. 4: Algorithm for Division

Subset0(P, x_k) becomes ($P\%x_k$). Change() operation can also be described using some multiplication and division operators. Using unate cube set expressions, we can elegantly express the algorithms or procedures for manipulating sets of combinations.

3.2 Algorithms of Operations

We show here that the above operations can be efficiently executed using 0-sup-BDD techniques. The three trivial cube sets are represented by simple 0-sup-BDDs. The empty set "0" becomes 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. The intersection, union, and difference operations are the same as the basic operations of the 0-sup-BDDs shown in Section 2. The other three operations, product, quotient, and remainder, are not included in the basic ones. We have developed the algorithms for computing these operations.

If we execute the multiplication and division operations by processing each cube one by one, the execution time will depends on the length of expressions. Such a procedure is impractical when we deal with very large number of cubes. We developed new recursive algorithms based on 0-sup-BDDs to efficiently calculate large size of expressions.

Our algorithms are based on the divide-and-conquer method. Suppose x is the highest-ordered literal, P and Q are then factored into two-part:

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

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

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

Each sub-product term can be computed recursively. The expressions are eventually broken down into trivial ones and the results are obtained. In the worst case, this algorithm would require an exponential number of recursive calls for the number of literals; however, we can accelerate them by using a hash-based cache which memorizes results of recent operations. By referring to the cache before every recursive call, we can avoid duplicate executions for equivalent subsets. Consequently, the execution time depends on the size of 0-sup-BDDs, not on the number of cubes and literals. This algorithm is shown in detail in Fig. 3.

The quotient is computed in the same recursive manner. Suppose x is a literal at the root-node in Q, and P_0, P_1, Q_0, Q_1 are the sub cube sets factored by x. (Notice that $Q_1 \neq 0$ since x appears in Q.) The quotient (P/Q) can be described as:

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

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 subsets. This algorithm is illustrated in Fig. 4. The remainder $(P\%Q)$ can be determined by calculating $P - P * (P/Q)$.

4 Implementation and Applications

Based on the techniques mentioned above, we developed a Unate Cube set Calculator (UCC). This program is an interpreter with a lexical and syntax parser for calculating unate cube set expressions and displaying the results in various formats. Our program allows up to 65,535 different literals. An example of execution is shown in Fig. 5.

We can define the cost for each literal, for use in computing the minimum-cost cube. After constructing 0-sup-BDDs, the minimum-cost cube can be found in a time proportional to the number of nodes in the graph, as using conventional BDDs[5].

Because UCC can generate huge 0-sup-BDDs with millions of nodes, limited only by memory capacity, we can manipulate large-scale and complicated expressions. Here we show several applications for the unate cube set calculator.

4.1 8-Queens Problem

The 8-queens problem is one example in which using unate cube set calculation is more efficient than using ordinary Boolean expressions.

First, we allocate 64 logic 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:

- Only one variable is "1" in a particular column.

```
***** Unate Cube set Calculator (Ver. 1.1) *****
ucc> symbol a(2) b(1) c(2) d(3) e(2)
ucc> F = (a + b) (c + d + e)
ucc> print F
 a c, a d, a e, b c, b d, b e
ucc> print .factor F
 ( a + b ) ( c + d + e )
ucc> print .matrix F
  1.1..
  1..1.
  1...1
  .11..
  .1.1.
  .1..1
ucc> print .count F
  6
ucc> print .size F
  5 (10)
ucc> G = F * a + c d e
ucc> print G
 a b c, a b d, a b e, a c, a d, a e, c d e
ucc> print .factor G
 a ( b + 1 ) ( c + d + e ) + c d e
ucc> print F & G
 a c, a d, a e
ucc> print F - G
 b c, b d, b e
ucc> print G - F
 a b c, a b d, a b e, c d e
ucc> print  G / (a b)
 c, d, e
ucc> print  G % (a b)
 a c, a d, a e, c d e
ucc> print .mincost G
 a c (4)
ucc> exit
```

Fig. 5: Execution of Unate Cube Set Calculator

- Only one variable is "1" in a particular row.
- One or no variable is "1" on a particular diagonal line.

To solve this problem using conventional BDDs based on Boolean algebra, we construct each column/row/diagonal constraint with a BDD. Then we compute conjunction for all the constraints to generate a BDD representing the whole solution.

Table 1 shows the experimental results of this procedure. We observed the number of BDD nodes and possible solutions at each intermediate steps, at after computing conjunction of column conditions (column), then after including row condition (row), after ascending diagonals (dia-asc), and finally after descending diagonals (final). This experiment shows that the number of solutions decreases monotonically during computation, but size of BDDs temporarily grows much larger than the final size. This temporary growth may cause overflow when solving larger-scale problems.

By unate cube set calculation, we can more efficiently solve the 8-queens problem in another way:

1. Search all the choices to put the first queen.
2. Search all the choices to put the second queen, considering the first queen's location.
   ...
8. Search all the choices to put the eighth queen, considering the other queens' locations.

Table 1: 8-Queens Using Boolean Algebra

Step	column	row	dia-asc	final
#Node	119	3330	10308	2450
#Solution	16777216	40320	2113	92

Table 2: 8-Queens Using Unate Cube Set Algebra

Step	1	2	3	4	5	6	7	8
#Node	8	35	107	246	504	715	647	373
#Solution	8	42	140	344	568	550	312	92

Table 3: Results on N-Queens Problems

N	Lit.	Sol.	BDD	ZBDD	(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) BDD/ZBDD, (Z/S) ZBDD/Solution.

This algorithm can be described with unate cube set expressions as:

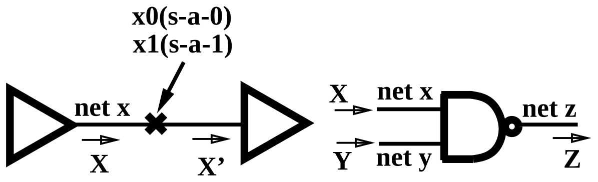
$$
\begin{aligned}
S_1 &= x_{11} + x_{12} + \ldots + x_{18} \\
S_2 &= x_{21}(S_1\%x_{11}\%x_{12}) + x_{22}(S_1\%x_{11}\%x_{12}\%x_{13}) \\
&\quad + \ldots + x_{28}(S_1\%x_{17}\%x_{18}) \\
S_3 &= x_{31}(S_2\%x_{11}\%x_{13}\%x_{21}\%x_{22}) \\
&\quad + x_{32}(S_2\%x_{12}\%x_{14}\%x_{21}\%x_{22}\%x_{23})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

Calculating these expressions with 0-sup-BDDs provides the set of solutions to the 8-queens problem. Table 2 shows the experimental results of our new approach. In this method, we can limit the growth in graph size, and thus, the final size of 0-sup-BDDs is much less than when using conventional BDDs.

H. Okuno reported experimental results for N-queens problems[8]. In Table 3, the column "BDD" shows the size of BDDs using Boolean algebra, and "ZBDD" is the size of 0-sup-BDDs using unate cube set algebra. This shows that there are about N times less 0-sup-BDDs than conventional BDDs. We can represent all the solutions at once within a storage space almost proportional to the number of solutions.

4.2 Fault Simulation

N. Takahashi et al. proposed a method of fault simulation for multiple faults using BDDs[9]. This is a deductive method for multiple faults, that manipulates sets of multiple stuck-at faults using BDDs. It propagates the fault sets from primary inputs to primary outputs, and eventually obtains the detectable faults at primary outputs. The study [9] used conventional BDDs, however; we can more



(a) At a Net (b) At a Gate

Fig. 6: Fault Set Propagation

simply compute the fault simulation using 0-sup-BDDs based on unate cube set algebra.

First, we generate the whole set of multiple faults that is assumed in the simulation. The set 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 similar. We can represent the set of double and single faults F_2 as $(F_1 * F_1)$. Further, $(F_2 * F_1)$ gives the set of three or less 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.

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. 6(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 expression as:
$X' = (X+(U/x_1)*x_1)\%x_0$, when $x = 0$ in a good circuit.
$X' = (X+(U/x_0)*x_0)\%x_1$, when $x = 1$ in a good circuit.
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. 6(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 expressions as follows:
$Z = X \,\&Y$, when $x = 0, y = 0, z = 1$ in a good circuit.
$Z = X - Y$, when $x = 0, y = 1, z = 1$ in a good circuit.
$Z = X + Y$, when $x = 1, y = 1, z = 0$ in a good circuit.
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 directly execute it by a unate cube set calculator.

5 Conclusion

We have discussed unate cube set algebra based on 0-sup-BDDs. We proposed efficient algorithms for computing unate cube set operations including multiplication and division. We have developed a unate cube set calculator, which can be applied to many practical problems.

Unate cube sets have different semantics from binate cube sets; however, there is a way to simulate binate cube sets using unate ones. We use two unate literals x_1 and x_0 for one binate literal. For example, a binate cube set $(a\,\bar{b} + c)$ is expressed as the unate cube set $(a_1 b_0 + c_1)$. In this way, We can easily simulate the cube-based algorithms implemented in the logic design systems such as ESPRESSO and MIS[7]. Utilizing this technique, we have developed a practical multi-level logic optimizer. It is detailed elsewhere[10].

Unate cube set expressions are suitable for representing sets of combinations, and they can be efficiently manipulated using 0-sup-BDDs. For solving some types of combinatorial problems, our method are more useful than using conventional BDDs based on Boolean algebra. We expect the unate cube set calculator to be utilized as a helpful tool in researching and developing LSI design systems.

Acknowledgment

The author wish to acknowledge the valuable discussion with H. Okuno and N. Takahashi.

References

[1] R. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Trans. Comput.*, Vol. C-35, No. 8, pp. 677-691, Aug.1986.

[2] Y. Matsunaga and M. Fujita, "Multi-level logic optimization using binary decision diagrams," *Proc. of ACM/IEEE ICCAD'89*, pp. 556-559, Nov. 1989.

[3] J. Burch, E. Clarke, K. McMillan and D. Dill, "Sequential circuit verification using symbolic model checking," *Proc. of ACM/IEEE DAC'90*, pp.618-624, June 1990.

[4] O. Coudert and J. Madre and H. Fraisse, "A new viewpoint of two-level logic optimization," *Proc. of ACM/IEEE DAC'93*, pp. 625-630, June 1993.

[5] B. Lin and F. Somenzi, "Minimization of symbolic relations," *Proc. of ACM/IEEE ICCAD'90*, pp. 88-91, 1990.

[6] S. Minato, "Zero-suppressed BDDs for set manipulation in combinatorial problems," *Proc. of ACM/IEEE DAC93*, pp.272-277, 1993.

[7] R. Brayton, R. Rudell, A. S.-Vincentelli and A. Wang, "MIS: a multiple-level logic optimization system," *IEEE Trans. CAD*, Vol. CAD-6, No. 6, pp. 1062-1081, June 1987.

[8] H. Okuno, "Reducing combinatorial explosions in solving search-type combinatorial problems with binary decision diagram" (*in Japanese*), *Trans. of Information Processing Society of Japan*, Vol.35, No.5, May 1994, in press.

[9] N. Takahashi, N. Ishiura and S. Yajima, "Fault simulation for multiple faults using shared BDD representation of fault sets," *Proc. of ACM/IEEE ICCAD'91*, pp. 550-553, Nov. 1991.

[10] S. Minato, "Fast weak-division method for implicit cube representation," *Proc. of the Synthesis and Simulation Meeting and International Interchange (SASIMI'93, Japan)*, pp. 423-432, Oct. 1993.