IEICE TRANS. FUNDAMENTALS, VOL. E76-A, NO. 6 JUNE 1993

PAPER

Fast Generation of Prime-Irredundant Covers from Binary Decision Diagrams

Shin-ichi MINATO†, *Member*

SUMMARY Manipulation of Boolean functions is one of the most important techniques for implementing of VLSI logic design systems. This paper presents a fast method for generating *prime-irredundant covers* from Binary Decision Diagrams (BDDs), which are efficient representation of Boolean functions. Prime-irredundant covers are forms in which each cube is a prime implicant and no cube can be eliminated. This new method generates compact cube sets from BDDs *directly*, in contrast to the conventional cube set reduction algorithms, which commonly manipulate redundant cube sets or truth tables. Our method is based on the idea of a *recursive operator*, proposed by Morreale. Morreale's algorithm is also based on cube set manipulation. We found that the algorithm can be improved and rearranged to fit BDD operations efficiently. The experimental results demonstrate that our method is efficient in terms of time and space. In practical time, we can generate cube sets consisting of more than 1,000,000 literals from multi-level logic circuits which have never previously been flattened into two-level logics. Our method is more than 10 times faster than *ESPRESSO* in large-scale examples. It gives quasi-minimum numbers of cubes and literals. This method should find many useful applications in logic design systems.

key words: *BDDs (Binary Decision Diagrams), prime-irredundant covers, Boolean functions, sum-of-products forms, logic synthesis*

1. Introduction

Manipulation of Boolean functions is one of the most important techniques for implementing VLSI logic design systems. Binary Decision Diagrams (BDDs), which were proposed by Akers[1] and Bryant,[2] are graph representations of Boolean functions. Recently, BDDs have attracted much attention because they enable us to manipulate Boolean functions efficiently in terms of time and space.[3],[4]

In many logic design systems, cube sets (also called covers, PLA forms, sum-of-products forms, or two-level logics) are employed to represent Boolean functions. Cube sets have been extensively studied for many years. Cube set manipulation algorithms will assume greater importance as time goes on. In general, it is not so difficult to generate BDDs from cube sets, but there are no efficient methods for generating compact cube sets from BDDs.

In this paper, we present a fast method of generating prime-irredundant covers from BDDs. In such covers, each cube is a prime implicant and no cube can be eliminated.

The minimization or optimization of cube sets has received much attention, and a number of efficient algorithms, such as *MINI*[6] and *ESPRESSO*[7] have been developed. Since these methods are based on the manipulation of cube sets or truth tables, they cannot be applied to BDD operations directly. Our method is based on the idea of the *recursive operator*, proposed by Morreale.[5] We found that Morreale's algorithm, which is also based on cube set manipulation, can be improved and efficiently adapted for BDD operations.

The features of our method are summarized as follows:

- Prime and irredundant representation can be obtained.
- It generates cube sets from BDDs *directly* without the temporary generation of redundant cube sets in the process.
- It can handle the *don't cares*.
- The algorithm can be extended to manage multiple output functions.

Experimental results show that our method is efficient in terms of time and space. In a practical time, we can generate cube sets consisting of more than 100,000 cubes and 1,000,000 literals from multi-level logic circuits which have never previously been flattened into two-level logics. Our method gives the quasi-minimum numbers of cubes and literals, but it does not always give the minimum ones.

The remainder of this paper is organized as follows. In Sect. 2, we explain the properties of our BDD package. In Sect. 3, we show conventional methods of generating cube sets from BDDs. In Sect. 4, we present the properties of prime-irredundant covers and describe our algorithm of generating prime-irredundant covers from BDDs. The experimental results are shown in Sect. 5, followed by conclusion in Sect. 6.

2. BDD Package

We begin with an explanation of our BDD package. BDDs are graph representations of Boolean functions, as shown in Fig. 1. They are graphs of binary

Manuscript received July 3, 1992.
Manuscript revised December 16, 1992.

† The author is with NTT LSI Laboratories, Atsugi-shi, 243-01 Japan.

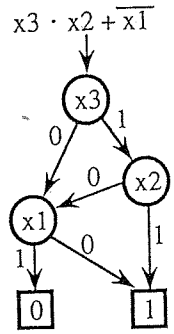


Fig. 1 A BDD

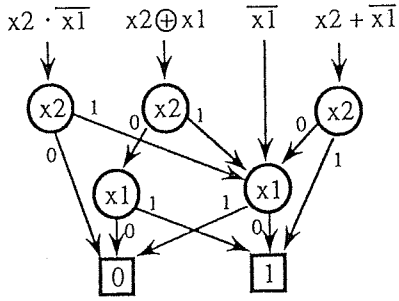


Fig. 2 A shared BDD.

decision trees representing recursive *Shannon's expansions* which are completely reduced by eliminating all redundant nodes and by sharing all the equivalent nodes. The reduced BDDs are characterized by the fact that each of them represents a Boolean function *uniquely* if the order of the input variables is fixed (see Refs. (1), (2) for details).

A set of BDDs representing multiple functions can be united into a single graph that consists of the BDDs with shared subgraphs. The efficiency of manipulation can be improved by managing all the BDDs that appear as a single graph, as in Fig. 2. We call such graphs SBDDs (Shared BDDs).[9] We can furthermore reduce the operation time and memory requirement by using *attributed edges*,[8],[9] which represent certain logic operations such as inverting.

Our BDD package, implemented with the above techniques, exhibits the following useful attributes.

- After generating BDDs, the equivalence of two functions can be checked in a constant time.
- Inverting operation can be executed in a constant time.
- Binary operations such as "and," "or" can be carried out within a time that is almost proportional to the size of the graphs.

The size of BDDs largely depends on the order of the input variables. It is difficult to derive a method that always yields the best order, but with some heuristic methods, we are able to find a tolerable order in many cases.[9]-[11]

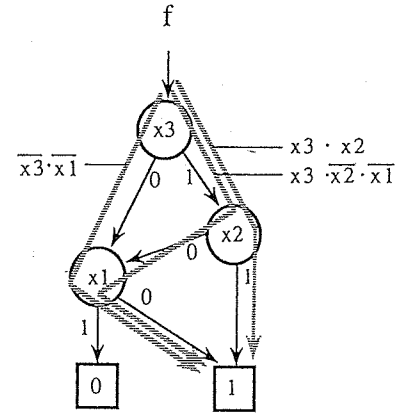


Fig. 3 1-path enumeration method.

3. Conventional Generation of Cube Sets from BDDs

Akers[1] presented a method of generating cube sets from BDDs by enumerating the "1" paths. This method enumerates all the paths from the root node toward the "1" leaf node and lists the cubes which represent the input values to activate such paths. In the example shown in Fig. 3, we can find the three paths which lead to the cube set:

$$x_3x_2 + x_3\bar{x}_2\bar{x}_1 + \bar{x}_3\bar{x}_1.$$

In reduced BDDs, all the redundant nodes are eliminated, thereby the literals of the eliminated nodes never appear in the cubes. In the above example, the first cube has neither x_1 nor $\bar{x}_1$. All of the cubes generated in this method are disjoint because no two paths can be activated at the same time.

This method is good for generating *disjoint covers*; however, it does not necessarily give the *minimum covers*. For example in evidence, the literal of the root node appears in every cube, but some of them may be needless. In general, they still have considerable redundancy in terms of the number of cubes or literals.

Recently, Jacobi and Trullemans[12] presented the method of removing such redundancy. The method generates a prime-irredundant cover from a BDD in a divide-and-conquer manner. On each node of the BDD, the method generates two cube sets for the two subgraphs of the node, and then combines the two by eliminating redundant literals and cubes.

In this method, a cube set is represented with a list of BDDs each of which represents a cube. Each cube is determined whether it is redundant or not by applying BDD operations. Their experiment shows the method is efficient in many cases. However, the method still has inefficiency that it generates redundant cubes temporarily and removes them during the process.

In the following sections, we propose a fast method of generating prime-irredundant covers from BDDs *directly*.

4. Generation of Prime-Irredundant Covers

In this section, we present the properties of prime-irredundant covers and describe the algorithm for generating prime-irredundant covers from BDDs directly.

4. 1 Prime-Irredundant Covers

If a cube set has the following two properties, we call a prime-irredundant cover.

- Each cube is a prime implicant; that is, no literal can be eliminated without changing the function.
- There are no redundant cubes. In other words, no cube can be eliminated without changing the function.

For example, the expression $xyz+x\bar{y}$ is not a prime-irredundant cover because we can eliminate a literal without changing the fuction. The expression $xz+\bar{x}\bar{y}$ is a prime-irredundant cover.

Prime-irredundant covers are very compact in general, but they are not necessarily the minimum form. The following three expressions represent the same function and all of them are prime and irredundant.

$$x\bar{y}+xz+\bar{x}y+\bar{x}\bar{z},\quad x\bar{y}+\bar{x}y+yz+\bar{y}\bar{z},$$

$$x\bar{y}+\bar{x}\bar{z}+yz$$

From this we can see that prime-irredundant covers as well as their size can vary. However, empirically they are not very different from the minimum form in terms of size.

Prime-irredundant covers are useful for many applications including logic synthesis, fault testable design, and combinatorial optimization problems.

4. 2 Morreale's Algorithm

Our method is based on the *recursive operator* proposed by Morreale.[5] His algorithm recursively deletes redundant cubes and literals from a given cube set. The basic idea is summarized in this expansion:

$$isop = \bar{v}\cdot isop_0 + v\cdot isop_1 + isop_*$$

where *isop* represents the cube set of a prime-irredundant cover, and v is one of the input variables. This expansion means that the cube set can be divided into three subsets containing $\bar{v}$, v, an the others. Then, excluding $\bar{v}$ and v from each cube, the three subsets of $isop_1$, $isop_0$ and $isop_*$ should also be prime-irredundant covers. Based on this expansion, the algorithm generates a prime-irredundant cover recursively (see Ref. (5) for details).

Unfortunately, Morreale's method is not efficient for large-scale functions because the algorithm is based on cube set representation and it takes a long time to manipulate cube sets for tautology checking, inverting, and other logic operations. However, the basic idea of "recursive expansion" is well suited to BDD manipula-

```
ISOP(f(x)) {
/* (input) f(x) : {0,1}^n → {0,1,*} */
/* (output) isop : prime-irredundant covers*/
  if ( ∀x ∈ {0,1}^n; f(x) ≠ 1 ) { isop ← 0 ; }
  else if ( ∀x ∈ {0,1}^n; f(x) ≠ 0 ) { isop ← 1 ; }
  else {
    v ← one of x ;
      /* v is the input with highest order in BDD */
    f0 ← f(x |v=0) ; /* the subfunction on v = 0 */
    f1 ← f(x |v=1) ; /* the subfunction on v = 1 */
    Compute f0', f1' in the following rules;
```

f0'	f1 \ f0	0	1	*
	0	0	1	*
	1	0	*	*
	*	0	*	*

f1'	f1 \ f0	0	1	*
	0	0	0	0
	1	1	*	*
	*	*	*	*

```
    isop0 ← ISOP(f0') ;
      /* recursively generates cubes including v̄ */
    isop1 ← ISOP(f1') ;
      /* recursively generates cubes including v */
    Let g0, g1 be the covers of isop0, isop1, respectively;
    Compute f0'', f1'' in the following rules;
```

f0''	g0 \ f0	0	1	*
	0	0	1	*
	1	–	*	*

f1''	g1 \ f1	0	1	*
	0	0	1	*
	1	–	*	*

```
    Compute f* in the following rule;
```

f*	f1'' \ f0''	0	1	*
	0	0	0	0
	1	0	1	1
	*	0	1	*

```
    isop* ← ISOP(f*) ;
      /* recursively generates cubes excluding v̄, v */
    isop ← v̄ · isop0 + v · isop1 + isop* ;
  }
  return isop ;
}
```

Fig. 4 Algorithm for generating prime-irredundant covers.

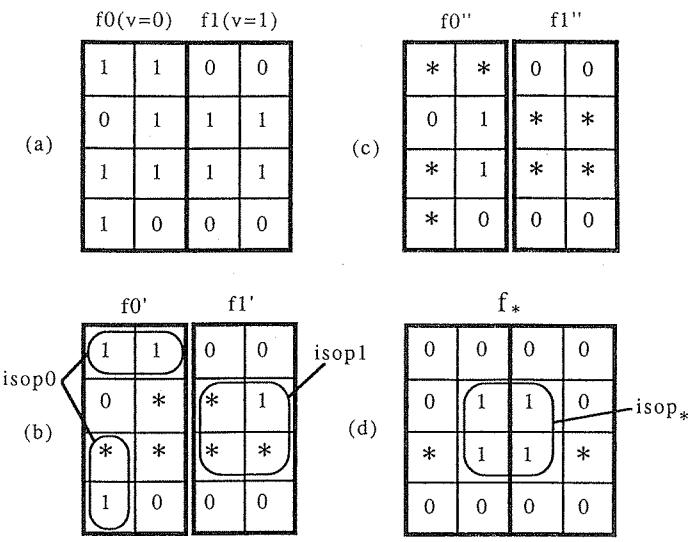


Fig. 5 An example.

tion, which is what motivated us to improve and adapt Morreale's method for BDD representation.

4.3 Algorithm Using BDDs

Our method generates a compact cube set from a BDD directly, not through redundant cube sets. The algorithm, called *ISOP* (*Irredundant Sum-Of-Poducts generation*), is described in Fig. 4. Here we illustrate how it works using the example shown in Figs. 5(a) through (d). In Fig. 5(a), the function f is divided into the two subfunctions, f_0 and f_1, by assigning 0, 1 to the input variable with the highest order in the BDD. In (b), f_0' an f_1' are derived from f_0 and f_1 by assigning a *don't care* value to the minterms which commonly give $f_0=1$ *and* $f_1=1$. f_0' and f_1' represent the minterms to be covered by the cubes including v or $\bar{v}$. We thereby generate their prime-irredundant covers $isop_0$ and $isop_1$ recursively. In Fig. 5(c), f_0'' and f_1'' are derived from f_0 and f_1 by assigning a *don't care* value to the minterms which are already covered by $isop_0$ or $isop_1$, and in (d) f_* is computed with f_0'' and f_1'' represents the minterms to be covered by the cubes excluding v and $\bar{v}$. We thereby generate its prime-irredundant cover $isop_*$ recursively. Finally, the result of *isop* can be obtained as the united set of $\bar{v}\cdot isop_0$, $v\cdot isop_1$ and $isop_*$.

Note that in practice the functions are represented and manipulated using BDDs. Here we employ Karnaugh maps to illustrate.

If the expanding order of the input variables is fixed, ISOP generates a unique form for each function, i.e., it gives a unique form for an ordered BDD. Another feature of this algorithm is that it can be applied for functions with don't cares.

This algorithm is well suited for BDD operations because:

- The subfunctions f_0 and f_1 can be derived from f in a constant time.
- Many redundant expansions can be avoided naturally because the redundant nodes are eliminated in reduced BDDs.
- Results can be returned quickly by checking the tautology of subfunctions and interrupting the recursive process.

It is difficult to exactly evaluate the time complexity of this algorithm. In our experiments, as will be shown later, the execution time was almost proportional to the product of the BDD size and the size of the cover generated.

4.4 Techniques for Implementation

In our method, ternary-valued functions including the *don't care* value are manipulated. Since our BDD package can only manage binary functions, we represent a ternary function with a pair of binary functions $(\lfloor f \rfloor, \lceil f \rceil)$, coded as

f	$\lfloor f \rfloor$	$\lceil f \rceil$
0	0	0
1	1	1
*	0	1

In this coding, the tautology of the function allowing the *don't care* value can be checked as $\lceil f \rceil \equiv 1$. The special operation for ternary-valued functions can be computed in the combination of ordinary logic operation for $\lfloor f \rfloor$ and $\lceil f \rceil$. For example, the ternary-valued operation:

f_0': f_1 \ f_0	0	1	*
0	0	1	*
1	0	*	*
*	0	*	*

can be written as:

$$(\lfloor f_0' \rfloor, \lceil f_0' \rceil) \leftarrow (\lfloor f_0 \rfloor \cdot \overline{\lceil f_1 \rceil}, \lceil f_0 \rceil).$$

We described earlier that *isop* is obtained as the union of the three cube sets, as shown in Fig. 4. In order to avoid cube set manipulation, we implemented the method in such a way that the results of cubes are directly dumped out to a file. On each recursive call, we push the processing literal to a stack, which we call a *cube stack*. When a tautology function is detected, the current content of the *cube stack* is appended to the output file as a cube. This approach is efficient because we can only manipulate BDDs, no matter how large the result of the cube set becomes.

Our method can be extended to manage multiple output functions. Sharing the common cubes among different outputs, we obtain more compact representation than if each output were processed separately. In our implementation, the cube sets of all the outputs are generated concurrently; that is, in Fig. 4 we extend f to be an array of BDDs as a multiple output function. Repeating recursive calls in the same manner as a single output function eventuates in the detection of a multiple output constant which consists of tautology outputs and inconsistency outputs. The tautology outputs means that those output functions include the cube which is currently kept in the *cube stack*.

5. Experimental Results

We implemented the method described in the foregoing section, and conducted some experiments to evaluate its performance. We used a SPARC Station 2 (SunOS 4.1.1, 32 MByte). The program is written in C and C++.

5.1 Comparison with ESPRESSO

First, we generated initial BDDs for the output functions of practical combinational circuits which may be multi-level or multiple output circuits. Then, we generated prime-irredundant covers from the BDDs and counted the numbers of the cubes and literals. We applied the *Dynamic Weight Assignment Method*[9] to find the proper order of the input variables for the initial BDD. The execution time includes the time to determine ordering, the formation of initial BDDs, and the time to generate prime-irredundant covers.

We compared our results with a conventional cube-based method. We flattened the given circuits into cube sets with the system MIS-II,[13] and then optimized the cube sets by ESPRESSO.[7]

The results are shown in Table 1. For circuits in this work we applied : an 8-bit data selector (dec8), an 8-bit priority encoder (enc8), a 4+4 bit adder (add4), an 8+8 bit adder (add8), a 2×2 bit multiplier (mult4) a 3×3 bit multiplier (mult6), a 24 input *Achilles' heel function*[7] (achil8p), and its complement (achil8n). Other items were chosen from benchmarks at MCNC'90.

The table shows that our method is much faster than ESPRESSO, with especially more than 10 times acceleration for the large scale circuits. The speed up was most impressive the c432 and c880, where we generated prime-irredundant covers consisting of more than 100,000 cubes and 1,000,000 literals within a reasonable time. We could not apply ESPRESSO to these circuits because we were unable to flatten them into cube sets even after ten hours. In another example, ESPRESSO performed poorly for achil8n because the *Achilles' heel function* requires a great many cubes when we invert it. Here, our method still puts in a good performance because the complementary function can be represented with the same size BDD as the original one.

As far as the number of cubes and literals is concerned, our method in general may give somewhat larger results than ESPRESSO. In most cases, the differences range between 0% and 20%. In none of experiments, did we find an example giving more than two times the difference in terms of number of literals.

Table 1 Comparison with ESPRESSO.

Name	In.	Out.	Our method			ESPRESSO		
			Cubes	Literals	Time(s)	Cubes	Lit.	Time(s)
dec8	12	2	17	90	0.3	17	90	0.2
enc8	9	4	17	56	0.2	15	51	0.3
add4	9	5	135	819	0.7	135	819	1.9
add8	17	9	2519	24211	13.3	2519	24211	443.1
mult4	8	8	145	945	1.4	130	874	5.0
mult6	12	12	2284	22274	26.7	1893	19340	1126.2
achil8p	24	1	8	32	0.2	8	32	2.0
achil8n	24	1	6561	59049	8.7	6561	59049	3512.7
5xp1	7	10	72	366	0.8	65	347	1.5
9sym	9	1	148	1036	0.9	87	609	10.7
alupla	25	5	2155	26734	20.5	2144	26632	257.3
bw	5	28	68	374	1.1	22	429	1.4
duke2	22	29	126	1296	3.2	87	1036	28.8
rd53	5	3	35	192	0.3	31	175	0.5
rd73	7	3	147	1024	1.2	127	903	4.2
sao2	10	4	76	575	1.1	58	495	2.4
vg2	25	8	110	914	1.9	110	914	42.8
c432	36	7	84235	969037	1744.8	×	×	>36k
c880	60	26	114299	1986014	1096.6	×	×	>36k

5.2 Effect of Variable Ordering

We conducted another experiment to evaluate the effect of variable ordering. In general, the size of BDDs depends greatly on the order. We generated prime-irredundant covers from the two BDDs of the same function but in a different order: one was in fairly good order (obtained using the *minimum-width method*[11]), and the other was in a random order.

As shown in Table 2, the numbers of cubes and literals are almost the same for both, while the size of BDDs varies greatly. The result demonstrates that our method is robust for variation in order. However, variable ordering is still important because it affects

Table 2 Effect of variable ordering.

Name	Heuristic order				Random order			
	#BDD	Cubes	Lit.	Time(s)	#BDD	Cubes	Lit.	Time(s)
dec8	16	17	90	0.3	41	17	90	0.3
enc8	21	17	56	0.2	25	17	56	0.2
add8	41	2519	24211	13.3	383	2519	24211	24.3
mult6	1274	2284	22274	26.7	1897	2354	22963	30.2
achil8n	24	6561	59049	8.7	771	6561	59049	30.9
5xp1	43	72	366	0.8	60	72	364	0.9
alupla	1376	2155	26734	20.4	4309	2155	26730	43.1
bw	85	68	374	1.1	90	64	353	1.1
duke2	396	126	1296	3.2	609	125	1280	3.7
sao2	143	76	575	1.1	133	76	571	1.0
vg2	108	110	914	1.9	1037	110	914	2.7

Table 3 Result for variation of input number. (100 random functions, single output)

In.	BDD size	Cubes	Literals	Lit./Cubes
1	0.58	0.77	1.35	1.75
2	1.41	1.25	2.84	2.27
3	3.22	2.30	7.17	3.12
4	6.39	4.20	16.05	3.82
5	11.71	7.85	36.39	4.64
6	20.51	14.88	82.18	5.52
7	36.24	27.09	172.06	6.35
8	64.59	52.27	377.41	7.22
9	118.17	99.31	808.09	8.14
10	210.12	192.26	1738.89	9.04
11	365.04	370.90	3693.49	9.96
12	633.97	722.11	7865.91	10.89
13	1144.12	1406.31	16635.79	11.83
14	2154.49	2752.53	35154.84	12.77
15	4151.45	5393.25	73980.57	13.72

the execution time and memory requirement.

5. 3 Statistical Properties

Taking advantage of our method, we examined the statistical properties of prime-irredundant covers. We applied our method to 100 patterns of random functions and took the average for the size of initial BDDs and generated cube sets. The random functions were computed using a standard C library.

Table 3 shows the property for variation in input number. Both BDDs and cube sets grow exponentially. It is known that the maximum BDD size is $O(2^n/n)$ (where n is the input number).[1] Our statistical experiment produced similar results. In terms of number of cubes, we observe about $O(2^n)$. The ratio of cubes to literals (the number of literals per cube) is almost proportional to n.

Table 4 shows the property for variation in output number when the input number is fixed. Both BDDs and cube sets grow a little less proportionally, appearing the effect of sharing their subgraphs or cubes. We expect such data sharing is more effective in practical circuits, since in many cases the output functions are relative each other, unlike random functions. The ratio of cubes to literals is almost constant, as the input number is fixed.

Table 4 Result for variation of output number. (input number=10)

Out.	BDD size	Cubes	Literals	Lit./Cubes
1	209.80	192.13	1737.84	9.05
2	364.44	381.69	3452.20	9.04
3	500.86	568.10	5145.01	9.06
4	630.93	754.88	6842.25	9.06
5	758.33	933.86	8468.70	9.07
6	884.87	1120.83	10166.36	9.07
7	1011.08	1294.84	11750.90	9.08
8	1136.94	1471.63	13355.59	9.08
9	1262.29	1649.47	14978.33	9.08
10	1388.76	1815.44	16493.02	9.08
11	1513.15	1987.56	18078.64	9.10

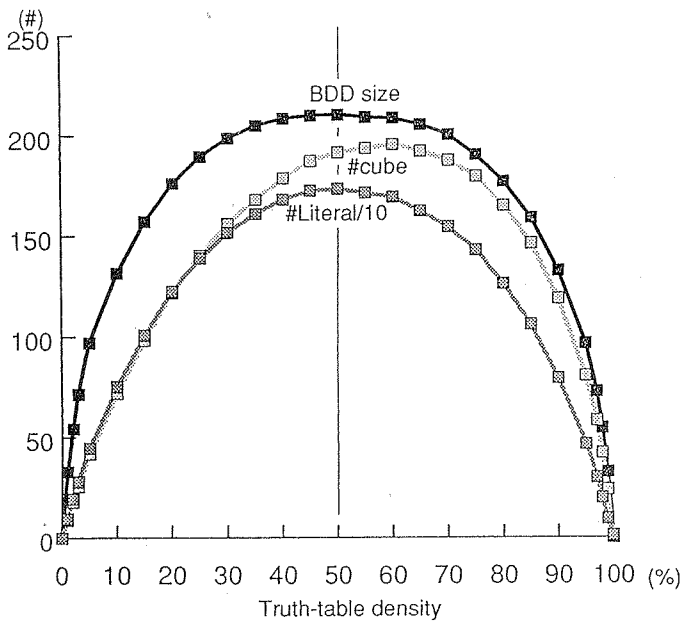


Fig. 6 Result for variation of truth-table density. (inputs=10, output=1)

Next, we investigated the property for variation in truth-table density, which is the rate of 1's in the truth table. We applied our method to the weighted random functions with 10 inputs, ranging from 0% to 100% in density. Figure 6 shows that the BDD size is symmetric with a center line at 50%, which is like the entropy of information. The number of cubes is not symmetric and peeks at about 60%; however, the number of literals becomes symmetric with BDD size. This result suggests that the number of literals is better as a measure of complexity of Boolean functions than the number of cubes.

6. Conclusion

We have presented a method of generating prime-irredundant covers from BDDs directly. The experiments show that our method is much faster than conventional methods. It enables us to generate compact cube sets from large-scale circuits, some of which have never been flattened into cube sets by previous methods. In terms of size of the result, our method may give somewhat larger results than ESPRESSO, but there are many applications in which such an increase is tolerable.

Our future work includes improving the method further more and applying to multi-level logic synthesis. Recently, *implicit cube representation*,[14] which is an compressed representation of cube sets, was presented. It allows us to deal with large-scale cube sets. We hope our method may be accelerated still more using the new representation.

Our method can be utilized to transform BDDs into compact cube sets or to flatten multi-level circuits into two-level circuits. We expect the method to be useful in many logic design system applications.

Acknowledgment

The author would like to express his appreciation to Mr. Adachi and Mr. Endo of NTT LSI laboratories for their encouragement. The author also thanks the members of Professor Yajima's research laboratory of Kyoto University for fruitful discussions.

References

(1) Akers, S. B., "Binary Decision Diagrams," *IEEE Trans. Comput.*, vol. C-27, no. 6, pp. 509-516, 1978.
(2) Bryant, R. E., "Graph-Based Algorithm for Boolean

Function Manipulation," *IEEE Trans. Comput.*, vol. C-35, no. 8, pp. 677-691, 1986.

(3) Minato, S., Ishiura, N. and Yajima, S., "Fast Tautology Checking Using Shared Binary Decision Diagram—Benchmark Results," *Proc. IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, pp. 580-584, 1989.

(4) Matsunaga, Y. and Fujita, M., "Multi-level Logic Optimization Using Binary Decision Diagrams," *Proc. ICCAD'89*, pp. 556-559, 1989.

(5) Morreale, E., "Recursive Operators for Prime Implicant and Irredundant Normal Form Determination," *IEEE Trans. Comput.*, vol. C-19, no. 6, pp. 504-509, 1970.

(6) Hong, S. J., Cain, R. G. and Ostapko, D. L., "MINI: A Heuristic Approach for Logic Minimization," *IBM Journal of Res. and Dev.*, vol. 18, no. 5, pp. 443-458, 1974.

(7) Brayton, R. K., McMullen, C. T., Hachtel, G. D., and S.-Vincentelli A. L., *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic Publishers USA, 1984.

(8) Madre, J. C. and Billon, J. P., "Proving Circuit Correctness Using Formal Comparison Between Expected and Extracted Behavior," *ACM/IEEE Proc. 25th DAC*, pp. 205-210, 1988.

(9) Minato, S., Ishiura, N. and Yajima, S., "Shared Binary Decision Diagram with Attributed Edges for Efficient Boolean Function Manipulation," *ACM/IEEE Proc. 27th DAC*, pp. 52-57, 1990.

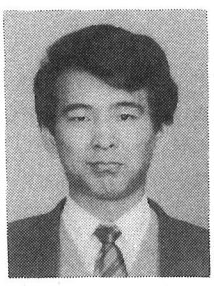
(10) Fujita, M., Matsunaga, Y. and Kakuda, T., "On Variable Ordering of Binary Decision Diagrams for the Application of Multi-level Logic Synthesis," *Proc. the European Conference on Design Automation*, pp. 50-54, 1991.

(11) Minato, S., "Minimum-Width Method of Variable Ordering for Binary Decision Diagrams," *IEICE Trans. Fundamentals*, vol. E75-A, no. 3, pp. 392-399, Mar. 1992.

(12) Jacobi, R. P. and Trullemans, A.-M., "Generating Prime and Irredundant Covers for Binary Decision Diagrams," *IEEE Proc. EDAC-92*, pp. 104-108, 1992.

(13) Brayton, R. K., Rudell, R., -Vincentelli, A. S. and Wang, A. R., "MIS: A Multiple-Level Logic Optimization System," *IEEE Trans. Comput. Aided Des. Integrated Circuits & Syst.*, vol. CAD-6, no. 6, pp. 1062-1081, 1987.

(14) Coudert, O. and Madre, J. C., "Implicit and Incremental Computation of Primes and Essential Primes of Boolean functions," *ACM/IEEE Proc. 29th DAC*, pp. 36-39, 1992.



Shin-ichi Minato was born in Ishikawa, Japan, on August 30, 1965. He received the B.E. and M.E. degrees in Information Science from Kyoto University, Japan in 1988 and 1990, respectively. Since joining NTT LSI Laboratories, Kanagawa, Japan in 1990, he has been working on the research of logic design systems. His current interest is in the representation and manipulation of Boolean functions for logic synthesis systems.