9

ARITHMETIC BOOLEAN EXPRESSIONS

9.1 INTRODUCTION

Boolean expressions are sometimes used in the research and development of digital systems, but calculating Boolean expressions by hand is a cumbersome job , even when they have only a few variables. For example, the Boolean expressions $(a\wedge\overline{b})\vee(\overline{a}\wedge\overline{c})\vee(b\wedge c)$, $(a\wedge\overline{b})\vee(a\wedge c)\vee(\overline{a}\wedge b)\vee(\overline{a}\wedge\overline{c})$, and $(a\wedge\overline{b})\vee(\overline{a}\wedge b)\vee(b\wedge c)\vee(\overline{b}\wedge\overline{c})$ represent the same function, but it is hard to verify their equivalence by hand. If they have more than five or six variables, we might as well give up. This problem motivated us to develop a Boolean expression manipulator (BEM)[MIY89], which is an interpreter that uses BDDs to calculate Boolean expressions. It enables us to check the equivalence and implications of Boolean expressions easily, and it helped us in developing VLSI design systems and solving combinatorial problems.

Most discrete problems can be described by logic expressions; however, the arithmetic operators such as addition, subtraction, multiplication, and comparison, are convenient for describing many practical problems. Such expressions can be rewritten using logic operators only, but this can result in expressions that are complicated and hard to read. In many cases, arithmetic operators provide simple descriptions of problems.

In this chapter, we present a new Boolean expression manipulator, that allows the use of arithmetic operators[Min93a]. This manipulator, called BEM-II, can directly solve problems represented by a set of equalities and inequalities, which are dealt with in 0-1 linear programming. Of course, it can also manipulate ordinary Boolean expressions. We developed several output formats for displaying expressions containing arithmetic operators.

In the following sections, we first show a method for manipulating Boolean expressions with arithmetic operations. We then present implementation of the BEM-II and its applications.

9.2 MANIPULATION OF ARITHMETIC BOOLEAN EXPRESSIONS

Most discrete problems can be described by using logic operations; however, arithmetic operators are useful for describing many practical problems. For example, a majority function with five inputs can be expressed concisely by using arithmetic operators:

$$x_1 + x_2 + x_3 + x_4 + x_5 \geq 3.$$

When, on the other hand, it is written using only Boolean expressions, this function becomes more complicated:

$$\begin{aligned}&(x_1 \wedge x_2 \wedge x_3) \vee (x_1 \wedge x_2 \wedge x_4) \vee (x_1 \wedge x_2 \wedge x_5)\\ &\vee (x_1 \wedge x_3 \wedge x_4) \vee (x_1 \wedge x_3 \wedge x_5) \vee (x_1 \wedge x_4 \wedge x_5)\\ &\vee (x_2 \wedge x_3 \wedge x_4) \vee (x_2 \wedge x_3 \wedge x_5) \vee (x_2 \wedge x_4 \wedge x_5)\\ &\vee (x_3 \wedge x_4 \wedge x_5).\end{aligned}$$

In this section, we describe an efficient method for representing and manipulating Boolean expressions containing arithmetic operators.

9.2.1 Definitions

We define *arithmetic Boolean expressions* and *Boolean-to-integer functions*, which are extended models of conventional Boolean expressions and Boolean functions.

Definition 9.1 Arithmetic Boolean expressions *are extended Boolean expressions that contain not only logic operators, but also arithmetic operators, such as addition (+), subtraction (−), and multiplication (×). Any integer can be used as a constant in the expression, but input variables are restricted to either 0 or 1. Equality (=) and inequalities (<, >, ≤, ≥, ≠) are defined as operations returning a value of either 1 (true) or 0 (false).* □

For example, $(3 \times x_1 + x_2)$ is an arithmetic Boolean expression with respect to the variables $x_1, x_2 \in \{0, 1\}$. $(3 \times x_1 + x_2 < 4)$ is also an example.

	x_1x_2 00	01	10	11
$3 \times x_1$	0	0	3	3
$3 \times x_1 + x_2$	0	1	3	4
$3 \times x_1 + x_2 < 4$	1	1	1	0

Figure 9.1 Computation of arithmetic Boolean expressions.

When ordinary logic operations are applied to integer values other than 0 and 1, we define them as bit-wise logic operations for binary-coded numbers, like the manner in many programming languages. For example, $(3 \vee 5)$ returns 7. Under this modeling scheme, conventional Boolean expressions become special cases of arithmetic Boolean functions.

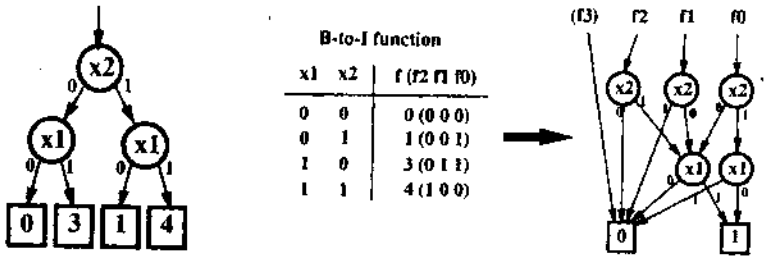
The value of the expression $(3 \times x_1 + x_2)$ becomes 0 when $x_1 = x_2 = 0$, and it becomes 4 when $x_1 = x_2 = 1$. We can see that an arithmetic Boolean expression represents a function from a binary-vector to an integer: $(B^n \rightarrow I)$. We call this a **Boolean-to-integer (B-to-I) function**, which has been discussed in Section 4.2. The operators in arithmetic Boolean expressions are defined as operations on B-to-I functions.

The procedure for obtaining the B-to-I function for the arithmetic Boolean expression $(3 \times x_1 + x_2 < 4)$ is shown in Fig. 9.1. First, multiply the constant function 3 by the input function x_1 to obtain the B-to-I function for $(3 \times x_1)$. Then add x_2 to obtain the function for $(3 \times x_1 + x_2)$. Finally we can get a B-to-I function for the entire expression $(3 \times x_1 + x_2 < 4)$ by applying the comparison operator $(<)$ to the constant function 4. We find that this arithmetic Boolean expression is equivalent to the expression $(\overline{x_1} \vee \overline{x_2})$.

9.2.2 Representation of B-to-I Functions

As shown in Fig. 9.1, a B-to-I function can be obtained by enumerating the output values for all possible combinations of the input values. But because this procedure is impracticable when there are many input variables (since the number of combinations grows exponentially), we need a more efficient way to represent B-to-I functions.

As discussed in Section 4.2, there are two ways to represent B-to-I functions: multi-terminal BDDs (MTBDDs) and BDD vectors. MTBDDs are extended BDDs with



(a) A MTBDD. (b) A BDD vector.

Figure 9.2 Representation for B-to-I functions.

multiple terminals, each of which has an integer value (Fig. 9.2(a)), and BDD vectors are the way to represent B-to-I functions with a number of BDDs by encoding the integer numbers into binary vectors (Fig. 9.2(b)).

The efficiency of the two representations depends on the nature of the objective functions. In manipulating arithmetic Boolean expressions, we often generate B-to-I functions from Boolean functions, as when we calculate $F \times 2$. $F \times 5$, or $F \times 100$ from a certain Boolean function F. In such cases, the BDD vectors can be conveniently shared with each other (Fig. 9.3). Multi-terminal BDDs, however, cannot be shared (Fig. 9.4), so we use BDD vectors for manipulating arithmetic Boolean expressions.

For negative numbers, we use 2's complement representation in our implementation. The most significant bit is used for the sign bit, whose BDD indicates the condition under which the B-to-I function produces a negative value. This coding scheme requires that the word-length be specified in order to know which is sign bit. An easy way is to allocate a long length in advance, but this limits the range of numbers. Our implementation supports a variable word-length for each B-to-I function, so there is no limit on the range of numbers.

9.2.3 Handling B-to-I functions

This section explains how to handle B-to-I functions represented by BDD vectors. Logic operations — such as AND, OR, and EXOR — are implemented as bit-wise operations between two BDD vectors. Applying BDD operations to their respective bits generates a new B-to-I function. We define two kinds of inversion operations:

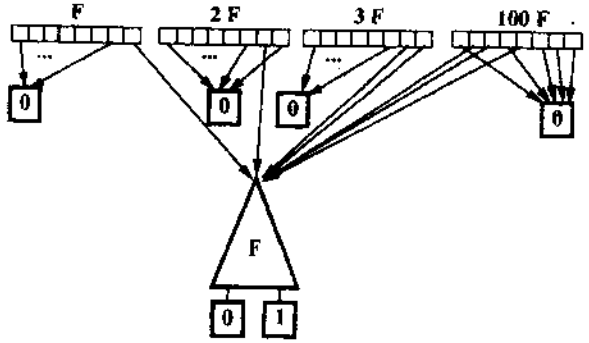


Figure 9.3 BDD vectors for arithmetic Boolean expressions.

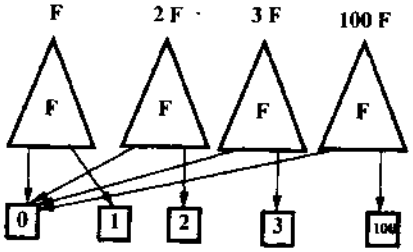


Figure 9.4 MTBDDs for arithmetic Boolean expressions.

bit-wise inversion and logical inversion. Logical inversion returns 1 for 0, and returns 0 for the other numbers.

Arithmetic addition can be composed using logic operations on BDDs by simulating a conventional hardware algorithm of full-adders designed as combinational circuits. We use a simple algorithm for a ripple carry adder, which computes from the lower bit to the higher bit, propagating carries. Other arithmetic operations like subtraction, multiplication, division, and shifting can be composed in the same way. Exception handling should be considered for overflow and division by zero.

Positive/negative checking is immediately performed by returning the sign-bit BDD. Using subtraction and sign checking, we can compose comparison operations between two B-to-I functions. These operations generate a new B-to-I function that returns a value of either 1 or 0 to express whether the equality or inequality is satisfied.

It would be useful to find the upper (or lower) bound value of a B-to-I function for all possible combinations of input values. This can be done efficiently by using a binary search. To find the upper bound, we first check whether the function can ever exceed 2^n. If there is a case in which it does, we then compare it with $2^n + 2^{n-1}$, otherwise only with 2^{n-1}. In this way, all the bits can be determined from the highest to the lowest, and eventually the upper bound is obtained. The lower bound is found in a similar way.

Computing the upper (or lower) bound is a unary operation for B-to-I functions; it returns a constant B-to-I function and can be used conveniently in arithmetic Boolean expressions. For example, the expression:

$$UpperBound(F) = F \quad (F \text{ is an arithmetic Boolean expression})$$

gives a function that returns 1 for the inputs that maximize F, otherwise it returns 0. That is, it computes the condition for maximizing F.

An example of calculating arithmetic Boolean expressions using BDD vectors is shown in Fig. 9.5.

9.2.4 Display Formats for B-to-I Functions

We propose several formats for displaying B-to-I functions represented by BDDs.

Integer Karnaugh Maps

A conventional Karnaugh map displays a Boolean function by using a matrix of logic values (0, 1). We extended the Karnaugh map to include integers as elements (Fig. 9.6). We call this *integer Karnaugh map*. It is useful for observing the behavior of B-to-I functions. Like ordinary Karnaugh maps, they are practical only for functions with fewer than five or six inputs. For a larger number of inputs, we can make an integer Karnaugh map with respect to only six input variables, by displaying the upper (or lower) bound for the rest of variables on each element of the map.

Bitwise Expressions

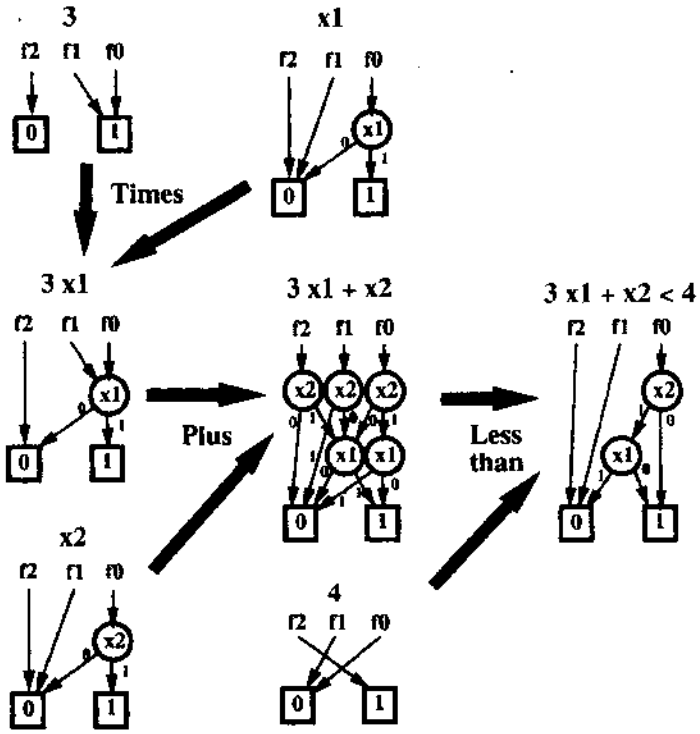


Figure 9.5 Generation of BDD vectors for arithmetic Boolean expressions.

When the objective function is too complicated for an integer Karnaugh map, the function can be displayed by listing Boolean expressions for respective bits of the BDD vector in the sum-of-products format. Figure 9.7 shows a bitwise expression for the same function shown in Fig .9.6. We used the ISOP algorithm (described in Chapter 5) to generate a compact sum-of-products format on each bit.

Bitwise expression is not very helpful for showing the behavior of B-to-I functions, but it does allow us to observe the appearance frequency of an input variable and it can estimate a kind of complexity of the functions.

F = 2 a + 3 b - 4 c + d

ab \ cd	00	01	11	10
00	0	1	-3	-4
01	3	4	0	-1
11	5	6	2	1
10	2	3	-1	-2

Figure 9.6 An integer Karnaugh map.

If a function never has negative values, we can suppress the expression for the sign bit. If some higher bits are always zero, we can omit showing them. With this zero suppression, a bitwise expression becomes a simple Boolean expression if the function returns only 1 or 0.

Case Enumeration

Using case enumeration, we can list all possible values of a function and display the condition for each case using a sum-of-products format (Fig. 9.8). This format is effective when there are many input variables but the range of output values is limited.

Arithmetic Sum-of-Products Format

It would be useful if we could display a B-to-I function as an expression using arithmetic operators. There is a trivial way of generating such an expression by using the case enumeration format. When the case enumeration method gives the values $v_1, v_2, \ldots, v_m$ and their conditions $F_1, F_2, \ldots, F_m$, we can create the expression $(v_1 \times F_1 + v_2 \times F_2 + \ldots + v_n \times F_m)$. In this method, the expression $(2 \times a + 3 \times b - 4 \times c + d)$ would thus be displayed as

$$\begin{aligned} &6 \times a\,b\,\overline{c}\,d + 5 \times a\,b\,\overline{c}\,\overline{d} + 4 \times \overline{a}\,b\,\overline{c}\,d + 3 \times (a\,\overline{b}\,\overline{c}\,d + \overline{a}\,b\,\overline{c}\,\overline{d}) \\ &+ 2 \times (a\,b\,c\,d + a\,\overline{b}\,\overline{c}\,\overline{d}) + (a\,b\,c\,\overline{d} + \overline{a}\,\overline{b}\,\overline{c}\,d) - (a\,\overline{b}\,c\,d + \overline{a}\,b\,c\,\overline{d}) \\ &- 2 \times a\,\overline{b}\,c\,d - 3 \times \overline{a}\,\overline{b}\,c\,d - 4 \times \overline{a}\,\overline{b}\,c\,\overline{d}. \end{aligned}$$

$$F = 2 \times a + 3 \times b - 4 \times c + d$$

$$\begin{aligned} \pm &: (\overline{a} \wedge c \wedge \overline{d}) \vee (\overline{b} \wedge c) \\ F_2 &: (a \wedge b \wedge \overline{c}) \vee (\overline{a} \wedge c \wedge \overline{d}) \vee (b \wedge \overline{c} \wedge d) \vee (\overline{b} \wedge c) \\ F_1 &: (a \wedge \overline{b}) \vee (a \wedge d) \vee (\overline{a} \wedge b \wedge \overline{d}) \\ F_0 &: (b \wedge \overline{d}) \vee (\overline{b} \wedge d) \end{aligned}$$

Figure 9.7 A bitwise expression

$$\begin{aligned} 6 &: a \wedge b \wedge \overline{c} \wedge d \\ 5 &: a \wedge b \wedge \overline{c} \wedge \overline{d} \\ 4 &: \overline{a} \wedge b \wedge \overline{c} \wedge d \\ 3 &: (a \wedge \overline{b} \wedge \overline{c} \wedge d) \vee (\overline{a} \wedge b \wedge \overline{c} \wedge \overline{d}) \\ 2 &: (a \wedge b \wedge c \wedge d) \vee (a \wedge \overline{b} \wedge \overline{c} \wedge \overline{d}) \\ 1 &: (a \wedge b \wedge c \wedge \overline{d}) \vee (\overline{a} \wedge \overline{b} \wedge \overline{c} \wedge d) \\ 0 &: (\overline{a} \wedge b \wedge c \wedge d) \vee (\overline{a} \wedge \overline{b} \wedge \overline{c} \wedge \overline{d}) \\ -1 &: (a \wedge \overline{b} \wedge c \wedge d) \vee (\overline{a} \wedge b \wedge c \wedge \overline{d}) \\ -2 &: a \wedge \overline{b} \wedge c \wedge \overline{d} \\ -3 &: \overline{a} \wedge \overline{b} \wedge c \wedge d \\ -4 &: \overline{a} \wedge \overline{b} \wedge c \wedge \overline{d} \end{aligned}$$

Figure 9.8 Case enumeration format.

This expression seems much more complicated than the original one, which has a linear form. Here we propose a method for eliminating the negative literals from the above expression and making an arithmetic sum-of-products expression which consists only of arithmetic addition, subtraction, and multiplication operators. Our method is based on the following expansion:

$$\begin{aligned} F &= x \times F_1 + \overline{x} \times F_0 \\ &= x \times (F_1 - F_0) + F_0, \end{aligned}$$

where F is the objective function, and F_0 and F_1 are subfunctions obtained by assigning 0 and 1 to input variable x. By recursively applying this expansion to all the input variables, we can generate an arithmetic sum-of-products expression containing no negative literals. We can thereby extract a linear expression from a B-to-I function represented by a BDD vector. For example, we can generate a BDD

vector for $2 \times (a + 3 \times b) - 4 \times (a + b)$, and it can be displayed in a reduced format as $(-2 \times a + 2 \times b)$.

The arithmetic sum-of-products format seems unsuitable for representing ordinary Boolean functions. For example, $(a \wedge \bar{b}) \vee (c \wedge \bar{d})$ becomes $-a\,b\,c\,d + a\,b\,c - a\,b + a\,c\,d - a\,c + a - c\,d + c$, which is more difficult to read than the original one.

9.3 APPLICATIONS

Using the techniques described above, we developed an arithmetic Boolean expression manipulator, called BEM-II. It is an interpreter with a lexical and syntax parser for calculating arithmetic Boolean expressions and displaying the results in various formats. This section gives the specifications of BEM-II and discusses some of their applications.

9.3.1 BEM-II Specification

BEM-II[1] has a C-shell-like interface, both for interactive execution from the keyboard and for batch jobs from a script file. It parses the script only from left to right; neither branches nor loop controls are supported. The available operators is listed in Table 9.1, and an execution example is shown in Fig. 9.9.

In BEM-II scripts, we can use two kind of variables: *input variables* and *register variables*. Input variables, denoted by strings starting with a lowercase letter, represent the inputs of the functions to be computed. They are assumed to have a value of either 1 or 0. Register variables, denoted by strings starting with an uppercase letter, are used to identify the memory to which a B-to-I function is to be saved temporarily. We can describe multi-level expressions using these two types of variables; for example,

```
F = a + b ; G = F + c.
```

Calculation results are displayed as expressions with input variables only, without using register variables. BEM-II allows 65,535 different input variables to be used, and there is no limit on the number of register variables.

BEM-II supports not only logical operators such as AND, OR, EXOR, and NOT, but also arithmetic operators such as addition, subtraction, multiplication, division, shifting, equality, inequality, and upper/lower bound. The syntax for expressions

[1] This program, which runs on a SPARC station, is a public domain software.
FTP server address: eda.kuec.kyoto-u.ac.jp (130.54.29.134) /pub/cad/BemII

Table 9.1 Operators in BEM-II.

(The upper operators are executed before the lower ones.)

( )
! (logical) ~ (bit-wise) + −(unary)
* /(quotient) %(remainder)
+ −(binary)
<< >> (bit-wise shift)
< <= > >= == != (relation)
& (bit-wise AND)
^ (bit-wise EXOR)
| (bit-wise OR)
? : (if-then-else)
UpperBound( ) LowerBound( )

generally conforms to C language specifications. The expression `A : B ? C` means *if-then-else*, and is equivalent to

```
((A != 0) * B) + ((A == 0) * C).
```

BEM-II generates BDD vectors of B-to-I functions for given arithmetic Boolean expressions. Since BEM-II can generate huge BDDs with millions of nodes, limited only by memory size, we can manipulate large-scale and complicated expressions. It can, of course, calculate any expressions that used to be manipulated by hand. The results can be displayed in the various formats presented in earlier sections.

The input variables are assumed to have values that are either 1 or 0, but multi-valued variables are sometimes used in real problems. In such cases, we can use register variables to deal with multi-valued variables. For example, `X = x0 + 2*x1 + 4*x2 + 8*x3` represents a variable ranging from 0 to 15. In another way, `X = x1 + 2*x2 + 3*x3 + 4*x4` represents a variable ranging from 1 to 4, under the one-hot constraint `(x1 + x2 + x3 + x4 == 1)`$_1$.

BEM-II can be used for solving many kind of combinatorial problems. Using BEM-II, we can generate BDDs for constraint functions of combinatorial problems specified by arithmetic Boolean expressions. This enables us to solve 0-1 linear programming problems by handling equalities and inequalities directly, without coding complicated procedures in a programming language. BEM-II can also solve problems which are expressed by non-linear expressions. BEM-II features its customizability: we can

```
% bemII
***** Arithmetic Boolean Expression Manipulator (Ver. 4.2) *****
> symbol a b c d
> F = 2*a + 3*b - 4*c + d
> print /map F
 a b : c d
      |    00    01    11    10
   00 |     0     1    -3    -4
   01 |     3     4     0    -1
   11 |     5     6     2     1
   10 |     2     3    -1    -2
> print /bit F
 F : !a & c & !d | !b & c
   2: a & b & !c | !a & c & !d | b & !c & d | !b & c
   1: a & !b | a & d | !a & b & !d
   0: b & !d | !b & d
> print F > 0
 a & b | a & !c | b & !c | !c & d
> M = UpperBound(F)
> print M
     6
> print F == M
 a & b & !c & d
> C = (F >= -1) & (F < 4)
> print C
 a & c & d | !a & !c & !d | b & c | !b & !c
> print /map C
 a b : c d
      |    00    01    11    10
   00 |     1     1     0     0
   01 |     1     0     1     1
   11 |     0     0     1     1
   10 |     1     1     1     0
> quit
%
```

Figure 9.9 An example of executing BEM-II.

compose scripts for various applications much more easily than we can by developing and tuning a specific program.

9.3.2 Subset-Sum Problem

The *subset-sum problem* is one example of a combinatorial problem that can easily be described by arithmetic Boolean expressions and solved by BEM-II. This problem is

to find a subset of a given set of positive integers $\{a_1, a_2, a_3, \ldots, a_n\}$, whose sum is a given number b. It is a fundamental problem for many applications, including VLSI CAD systems.

In BEM-II script, we use n input variables for representing whether or not the i-th number is chosen, and the constraint on these input variables can be described with simple arithmetic Boolean expressions. The following is an example of BEM-II script for a subset-sum problem:

```
symbol x1 x2 x3 x4 x5
S = 2*x1 + 3*x2 + 3*x3 + 4*x4 - 5*x5
C = (S == 12)
```

The `C` represents the assignments of the input variables for satisfying the constraint. This expression is almost the same as the definition of the problem. We can easily write and understand this script.

BEM-II is convenient not only for solving the problem but also for analyzing the nature of the problem. We can analyze the behavior of the constraint functions by displaying with various formats. An example of BEM-II execution for a subset-sum problem is shown in Fig. 9.10.

9.3.3 8-Queens Problem

8-queens problem is another good example showing that BEM-II is very convenient to describe the problems.

We first allocate 64 input variables corresponding to the squares on a chessboard. These represent whether or not there is a queen on that square. The constraints that the input variables should satisfy are expressed as follows:

- The sum of eight variables in the same column is 1.
- The sum of eight variables in the same row is 1.
- The sum of variables on the same diagonal line is less than 2.

These constraints can be described with simple arithmetic Boolean expressions as:

```
F1 = (x11 + x12 + x13 + ... + x18 == 1 )
```

```
% bemII
***** Arithmetic Boolean Expression Manipulator (Ver. 4.2) *****
> symbol x1 x2 x3 x4 x5
> S = 2*x1 + 3*x2 + 3*x3 + 4*x4 + 5*x5
> print /map S
 x1 x2 : x3 x4 x5
      |   000   001   011   010 |   110   111   101   100
   00 |     0     5     9     4 |     7    12     8     3
   01 |     3     8    12     7 |    10    15    11     6
   11 |     5    10    14     9 |    12    17    13     8
   10 |     2     7    11     6 |     9    14    10     5
> C = (S == 12)
> print C
 x1 & x2 & x3 & x4 & !x5 | !x1 & x2 & !x3 & x4 & x5 |
      !x1 & !x2 & x3 & x4 & x5
> print /map C ? S : 0
 x1 x2 : x3 x4 x5
      |   000   001   011   010 |   110   111   101   100
   00 |     0     0     0     0 |     0    12     0     0
   01 |     0     0    12     0 |     0     0     0     0
   11 |     0     0     0     0 |    12     0     0     0
   10 |     0     0     0     0 |     0     0     0     0
> print /map (S >= 12)? S : 0
 x1 x2 : x3 x4 x5
      |   000   001   011   010 |   110   111   101   100
   00 |     0     0     0     0 |     0    12     0     0
   01 |     0     0    12     0 |     0    15     0     0
   11 |     0     0    14     0 |    12    17    13     0
   10 |     0     0     0     0 |     0    14     0     0
> quit
%
```

Figure 9.10 Execution of BEM-II for a subset-sum problem.

```
F2 = (x21 + x22 + x23 + ... + x28 == 1 )
...
C = F1 & F2 & ...
```

BEM-II analyzes the above expressions directly. This is much easier than creating a specific program in a programming language. The script for the 8-queens problem took only ten minutes to create.

Table 9.2 shows the results when we applied this method to N-queens problems. In our experiments, we solved the problem up to $N = 11$. When seeking only one solution, we can solve the problem for larger values of N by using a conventional algorithm based on backtracking. However, the conventional method does not enumerate all the

Table 9.2 Results for N-queens problems.

N	Variables	BDD nodes	Solutions	Time(s)
8	64	2450	92	6.1
9	81	9556	352	18.3
10	100	25944	724	68.8
11	121	94821	2680	1081.9

solutions nor does it count the number of solutions for larger Ns. The BDD-based method generates all the solutions simultaneously and keeps them in a BDD. Therefore, if an additional constraint is appended later, we can revise the script easily without rewriting the program from the beginning. This customizability makes BEM-II very efficient in terms of the total time for programming and execution.

9.3.4 Traveling Salesman Problem

Traveling salesman problem (TSP) can also be solved by using BEM-II. The problem is to find the minimum-cost path for visiting all given cities once and returning to the starting city.

If n is the total number of cities, we allocate $n(n-1)/2$ input variables from x_{12} to $x_{(n-1)n}$, where x_{ij} represents the path between i-th city and j-th city. Using this variable scheme, we can express the constraints as follows.

- Each city has two path (coming in and going out):
  $x_{12} + x_{13} + x_{14} + \ldots + x_{1n} = 2$
  $x_{12} + x_{23} + x_{24} + \ldots + x_{2n} = 2$
  ...
  $x_{1n} + x_{2n} + \ldots + x_{(n-1)n} = 2$
- All the cities are connected:

  step-1
  Let $F_1 = 1, \quad F_2, F_3, \ldots, F_n = 0$.

  Repeat step-2 n times.

  step-2
  Let $F_1 = x_{12}F_2 \vee x_{13}F_3 \vee \ldots \vee x_{1n}F_n$.

Table 9.3 Results for traveling salesman problems.

n	Solutions	BDD nodes	Time(s)
8	2520	2054	8.7
9	66136	20160	28.8
10	181440	19972	216.5

Let $F_2 = x_{12}F_1 \vee x_{23}F_3 \vee \ldots \vee x_{2n}F_n$.

...

Let $F_n = x_{1n}F_1 \vee x_{2n}F_2 \vee \ldots \vee x_{(n-1)n}F_{n-1}$.

step-3 Condition $C = F_1 \wedge F_2 \wedge \ldots \wedge F_n$.

The logical product of all the above constraint expressions becomes the solution. BEM-II feeds these expressions directly, and generates BDDs representing all the possible paths to visit n cities. As mentioned in previous section, we can specify the cost (distance) of each path, and find an optimal solution to the problem after generating BDDs. Experimental results are listed in Table 9.3. We can solve the problem for n up to 10. This seems poor because the conventional method solves the problem when n is more than 1000. However, our method computes all the solutions at once, and the additional constraints can be specified flexibly. For example,

- There is a path that should be used, or should not be used.
- There is a city that should be visited first (second, third, ...).
- The starting city and the ending city are different.

These constraints can easily be expressed by arithmetic Boolean expressions, and BEM-II feeds them directly to solve the modified problems. It is not too late to develop the application-specific program after trying BEM-II.

9.3.5 Timing Analysis for Logic Circuits

In designing high-speed digital systems, timing analysis for logic circuits is important. The orthodox approach is to traverse the circuit to find the active path with the topologically maximum length. Takahara[Tak93] proposed a new method using

Table 9.4 Results of timing analysis.

Circuit	In.	Out.	Gates	Number of BDD nodes	
				Timing data	Logic data
cm138a	6	8	29	235	129
sel8	12	2	43	926	268
alu2	10	6	434	16,883	4,076
alu4	14	8	809	97,318	9,326
alupla	25	5	114	22,659	2,889
mult6	12	12	411	57,777	9,496
too_large	39	3	1044	730,076	10,789
C432	36	7	255	1,689,576	10,827

BEM-II. This method calculates B-to-I functions representing the delay time with respect to the values of the primary inputs. Using this method, we can completely analyze the timing behavior of a circuit for any combination of input values.

The B-to-I functions for the delay time can be described by a number of arithmetic Boolean expressions, each of which specifies the signal propagation at each gate. For a two-input AND gate with delay D, for example, if T_a and T_b are the signal arrival times at the two input pins, and V_a and V_b are their final logic values, the signal arrival time at output pin T_c is expressed as:

$T_c = T_b + D$ when $(T_a \leq T_b)$ and $(V_a = 1)$,
$T_c = T_a + D$ when $(T_a \leq T_b)$ and $(V_a = 0)$,
$T_c = T_a + D$ when $(T_a > T_b)$ and $(V_b = 1)$,
$T_c = T_b + D$ when $(T_a > T_b)$ and $(V_b = 0)$.

These rules can be described by the following arithmetic Boolean expression:

```
Tc = D + ((Ta > Tb)? (Vb? Ta:Tb):(Va? Tb:Ta)).
```

By calculating such expressions for all the gates in the circuit, we can generate BDD vectors for the B-to-I functions of the delay time. Experimental results for practical benchmark circuits are listed in Table 9.4[Tak93]. The size of the BDDs for the delay time is about 20 times greater than that of the BDDs for the Boolean functions of the circuits.

The generated BDDs maintain the timing information for all of the internal nets in the circuit, and we can use BEM-II to analyze the circuits in various ways. For example, we can easily compare the delay times of two nets in the circuit.

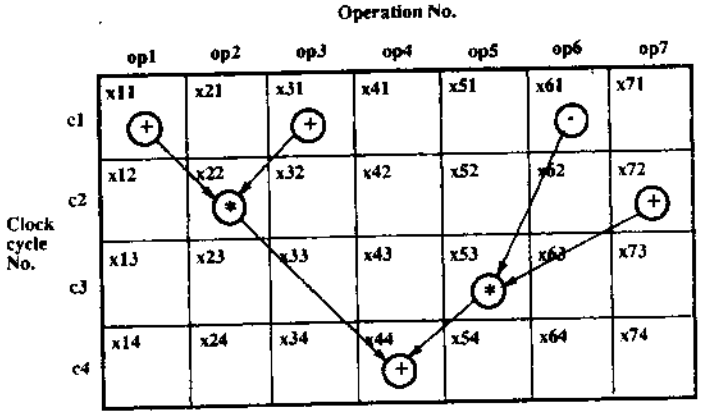


Figure 9.11 Example of a data-flow graph.

9.3.6 Scheduling Problem in Data Path Synthesis

Scheduling is one of the most important subtasks that must be solved to perform data path synthesis. Miyazaki[Miy92] proposed a method for solving scheduling problems by using BEM-II. The problem is to find the minimum-cost scheduling for a procedure specified by a data-flow graph under such constraints as the number of operation units and the maximum number of clock cycles (Fig. 9.11). This problem can be solved by using linear programming, but BEM-II can also be used.

When m is the total number of operations that appear in the data-flow graph and n is the maximum number of clock cycles, we allocate $m \times n$ input variables from x_{11} to x_{mn}, where x_{ij} represents the i-th operation executed in the j-th clock cycle. Using this variable coding, we can represent the constraints of the scheduling problem as follows.

1. Each operation has to be executed once:
   $x_{11} + x_{12} + \ldots + x_{1n} = 1$
   $x_{21} + x_{22} + \ldots + x_{2n} = 1$
   $\ldots$
   $x_{m1} + x_{m2} + \ldots + x_{mn} = 1$

Table 9.5 Results of scheduling problem.

Data	All solutions	Optimal solusions	c-step	Multi-plier	ALUs	BDD nodes	Time (sec)
DiffEq	108	3	4	2	2	321	2.1
Tseng	12	4	5	0	3	321	1.3
EWF	4200	167	14	2	3	1261	27.0

2. The same kind of operations cannot be executed simultaneously beyond the number of operation units. For example, when there are two adders and the a-th, b-th, and c-th operations require an adder:
   $x_{a1} + x_{b1} + x_{c1} \leq 2$
   $x_{a2} + x_{b2} + x_{c2} \leq 2$
   $\ldots$
   $x_{an} + x_{bn} + x_{cn} \leq 2.$

3. If two operations have a dependency in the data-flow graph, the operation in the upper stream has to be executed before the one in the lower stream.
   Let $C_1 = 1 \times x_{11} + 2 \times x_{12} + \ldots + n \times x_{1n}$.
   Let $C_2 = 1 \times x_{21} + 2 \times x_{22} + \ldots + n \times x_{2n}$.
   $\ldots$
   Let $C_m = 1 \times x_{m1} + 2 \times x_{m2} + \ldots + n \times x_{mn}$.
   Then $(C_i < C_j)$ is the condition that the i-th operation is executed before j-th one.

The logical product of all these constraint expressions becomes the solution to the scheduling problem. Using BEM-II, we can easily specify the cost of operation and the other constraints. BEM-II analyzes the above expressions and tries to generate BDDs that represent the solutions. If it succeeds in generating BDDs in main memory, we can immediately find a solution to the problem and count the number of solutions. Table 9.5[Miy92] lists the experimental results for benchmark data from the High-Level Synthesis Workshop (HLSW). The BDDs for constraint functions can be generated in a feasible memory and space.

9.4 CONCLUSION

We have developed an arithmetic Boolean expression manipulator (BEM-II) that can easily solve many kind of combinatorial problems by using arithmetic Boolean expressions. This manipulator can directly computes the equalities and inequalities in the constraints and costs of the problem, and generates BDDs that represent the solutions. It is therefore not necessary to write a specific program in a programming language for solving a problem. In addition to being used in the examples we have shown here, BEM-II can also be used for solving minimum-tree problems, magic squares, crypt-arithmetic problems, etc. Although the computation speed of BEM-II is not as great as that of well-optimized heuristic algorithms for large-scale problems, the customizability of BEM-II makes it very efficient in terms of total time for programming and execution. We expect it to be a useful tool in digital systems research and development.

10

CONCLUSIONS

In this book, we have discussed techniques related to BDDs and their applications for VLSI CAD systems.

In Chapter 2, we presented basic algorithms of Boolean function manipulation using BDDs. We then described implementation techniques to make BDD manipulators applicable to practical problems. As an improvement of BDDs, we proposed the use of *attributed edges*, which are edges attached with several sorts of attributes representing a certain operation. The *negative edge* is particularly effective and now widely used. We implemented a BDD subroutine package for Boolean function manipulation. It can efficiently represent and manipulate very large-scale BDDs containing more than a million of nodes. These techniques have been developed and improved in many laboratories all over the world[Bry86, MIY90, MB88, BRB90], and some program packages are open to the public. They are now widely utilized in research and development on VLSI CAD and other aspects of computer science.

The variable ordering method is one of the most important issues in the application of BDDs. In Chapter 3, we discussed the properties of variable ordering and showed two heuristic methods: *DWA method* and *minimum-width method*. The DWA method finds an appropriate order before generating BDDs. It refers to topological information which specifies the sequence of logic operations in the Boolean expression or logic circuit. Experimental results show that for many practical circuits the DWA method finds a tolerable order in a short computation time. The minimum-width method, on the other hand, finds an appropriate order for a given BDD without using additional information. It seeks a good order from a global viewpoint, rather than by doing an incremental search. In many cases, this method gives a better order than the DWA method does in a longer but still reasonable computation time. The techniques of variable ordering are still being studied intensively, but it is almost impossible to have

a method that always finds best order within a practical time. We will therefore have to use some heuristic methods selected according to the application.

In many applications, we sometimes use ternary-valued functions containing *don't cares*. The technique of handling *don't cares* are basic and important for Boolean function manipulation. In Chapter 4, we showed two methods — ternary-valued BDDs and BDD pairs — that we compared by introducing the D-variable. In this discussion, we showed that both the two methods are the special forms of the BDDs using the D-variable and that we can compare the efficiency of the two methods by considering of the properties of variable ordering.

We extended the technique of handling *don't cares* to be used for Binary-to-integer functions, and we presented two methods: MTBDDs and BDD vectors. These method can be compared by introducing the *bit-selection variables*, similarly to the D-variable for the ternary-valued functions. Multi-valued logic manipulation is important to broaden the scope of BDD application. These techniques are useful in various areas of computer science as well as for VLSI CAD.

In Chapter 5, we discussed the topic of transforming a BDD representation into another kind of data structure. We presented the *ISOP algorithm* that generates prime-irredundant cube sets directly from given BDDs, in contrast to the conventional cube set reduction algorithms, which temporarily generate redundant cube sets or truth tables. 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 that have never before been flattened into two-level logics. In experiments with large-scale examples, our method is more than 10 times faster than conventional methods. It gives quasi-minimum numbers of cubes and literals. In terms of size of the result, the ISOP algorithm may give somewhat larger results than ESPRESSO, but there are many applications in which such an increase is tolerable. Our method can be utilized to transform BDDs into compact cube sets or to flatten multi-level circuits into two-level circuits.

As our understanding of BDDs 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*. In Chapter 6, we proposed *Zero-Suppressed BDDs (ZBDDs)*, which are BDDs based on a new reduction rule. ZBDDs enable us to manipulate sets of combinations more efficiently than we can when using conventional BDDs. The effect of ZBDDs is especially remarkable when we are manipulating sparse combinations. We discussed the properties of ZBDDs and their efficiency as revealed by a statistical experiment. We then presented the basic operators for ZBDDs. These operators are defined as the operations on sets of combinations, and differ slightly from the operations on Boolean functions based on conventional BDDs.

On the basis of these ZBDD techniques, we discussed the calculation of unate cube set algebra. We developed efficient algorithms for computing unate cube set operations including multiplication and division, and we showed some practical applications. For solving some types of combinatorial problems, unate cube set algebra is more useful than using conventional logic operations. We expect the unate cube set calculator to be a helpful tool in developing VLSI CAD systems and in various other applications.

In Chapter 7, we presented an application for VLSI logic synthesis. We proposed a fast factorization method for cube set representation based on ZBDDs. Our new algorithm can be executed in a time almost proportional to the number of ZBDD nodes, which is usually much smaller than the number of literals in the cube set. By using this method, we can quickly generate multi-level logics from implicit cube sets even for parity functions and full-adders, which have never been practicable before. We implemented a new multi-level logic synthesizer, and experimental results indicate that our method is much faster than conventional methods, and the difference is remarkable when large cube sets are factorized. Our method greatly accelerates multi-level logic synthesis systems and makes them applicable to larger circuits. There is still some room to improve the results. We have used a simple strategy for choosing divisors, but more sophisticated strategies might be possible. Moreover, a Boolean division method for implicit cube sets is worth investigating to improve the optimization quality.

In Chapter 8, we discussed a ZBDD-based method of manipulating polynomials. We proposed an elegant way to represent polynomials by using ZBDDs, and showed efficient algorithms for operations on those representations. Our experimental results indicate that we can manipulate large-scale polynomials implicitly within a feasible time and space. An important feature of our representation is that it is the canonical form of a polynomial under a fixed variable order. As the polynomial calculus is a basic part of mathematics, our method is very useful in various areas.

In Chapter 9, we described a useful tool for the research on computer science. Using the BDD techniques, we have developed an arithmetic Boolean expression manipulator (BEM-II) that can easily solve many kind of combinatorial problems. It calculates not only binary logic operation but also arithmetic operations on multi-valued logics: such as addition, subtraction, multiplication, division, equality, and inequality. Such arithmetic operations provide simple descriptions for various problems. BEM-II feeds and computes the problems described as equalities and inequalities, which are dealt with using 0-1 linear programming. It is therefore not necessary to write a specific program in a programming language for solving a problem. In this chapter, we discussed the data structure and algorithms for the arithmetic operations. We then presented the specification of BEM-II and some application examples. Experimental results indicate that the customizability of BEM-II makes it very efficient in terms of

total time for programming and execution. We therefore expect it to be a useful tool for in digital systems research and development.

A number of works based on the BDD techniques are in progress. The novelty of BDD manipulation are summarized as:

1. Extracts the redundancy contained in the Boolean functions by using a fixed variable order.
2. Eliminates redundancy by ensuring that there are no duplicate nodes and that equivalent computation is not repeated.

The BDD algorithms are based on the quick search of the hash tables and the linked list data structure, both of which greatly benefit from the *random access machine model*,such that any data in the main memory can be accessed in a constant time. Because most computers are designed according to this model, we conclude that the BDD manipulation algorithms are fairly sophisticated and adapted to the conventional computer model. BDDs and their improvements will become key techniques in VLSI CAD systems and other aspects of computer science.

REFERENCES

[Ake78] S. B. Akers. Binary decision diagram. *IEEE Trans. on Computers*, Vol. C-27, No. 6, pp. 509–516, June 1978.

[BC95] R. E. Bryant and Y.-A. Chen. Verification of arithmetic circuits with binary moment diagrams. In *Proc. of 32nd ACM/IEEE Design Automation Conference (DAC'95)*, pp. 535–541, June 1995.

[BCMD90] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Sequential circuit verification using symbolic model checking. In *Proc. of 27th ACM/IEEE Design Automation Conference (DAC'90)*, pp. 46–51, June 1990.

[BF85] F. Brglez and H. Fujiwara. A neutral netlist of 10 combinational circuits. In *Proc. of 1985 IEEE International Symposium Circuit and Systems (ISCAS'85), Special Session on ATPG and Fault Simulation*, June 1985.

[BFG+93] R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic decision diagrams and their applications. In *Proc. of IEEE/ACM International Conference on Computer-Aided Design (ICCAD-93)*, pp. 188–191, November 1993.

[BHMSV84] R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. L. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers USA, 1984.

[BRB90] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a BDD package. In *Proc. of 27th ACM/IEEE Design Automation Conference (DAC'90)*, pp. 40–45, June 1990.

[BRKM91] K. M. Butler, D. E. Ross, R. Kapur, and M. R. Mercer. Heuristics to compute variable orderings for efficient manipulation of ordered binary decision diagrams. In *Proc. of 28th ACM/IEEE Design Automation Conference (DAC'91)*, pp. 417–420, June 1991.

[Bry86] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. on Computers*, Vol. C-35, No. 8, pp. 677–691, August 1986.

[Bry91] R. E. Bryant. On the complexity of VLSI implementations and graph representations of Boolean functions with application to integer multiplication. *IEEE Trans. on Computers*, Vol. C-40, No. 2, pp. 205–213, February 1991.

[BSVW87] R. K. Brayton, R. R. 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.

[CB89] K. Cho and R. E. Bryant. Test pattern generation for sequential MOS circuits by symbolic fault simulation. In *Proc. of 26th ACM/IEEE Design Automation Conference (DAC'89)*, pp. 418–423, June 1989.

[CHJ+90] H. Cho, G. D. Hachtel, S.-W. Jeong, B. Plessier, E. Schwarz, and F. Somenzi. ATPG aspects of FSM verification. In *Proc. of IEEE/ACM International Conference on Computer-Aided Design (ICCAD-90)*, pp. 134–137, November 1990.

[CM92] O. Coudert and J. C. Madre. A new graph based prime computation technique. In T. Sasao, editor, *New Trends in Logic Synthesis*, chapter 2, pp. 33–57. Kluwer Academic Publishers USA, 1992.

[CMF93] O. Coudert, J. C. Madre, and H. Fraisse. A new viewpoint of two-level logic optimization. In *Proc. of 30th ACM/IEEE Design Automation Conference (DAC'93)*, pp. 625–630, June 1993.

[CMZ+93] E. M. Clarke, K. L. McMillan, X. Zhao, M. Fujita, and J. Yang. Spectral transforms for large Boolean functions with applications to technology mapping. In *Proc. of 30th ACM/IEEE Design Automation Conference (DAC'93)*, pp. 54–60, June 1993.

[dG86] A. J. de Geus. Logic synthesis and optimization benchmarks for the 1986 DAC. In *Proc. of 23rd ACM/IEEE Design Automation Conference (DAC'86)*, pp. 78, June 1986.

[FFK88] M. Fujita, H. Fujisawa, and N. Kawato. Evaluation and improvement of Boolean comparison method based on binary decision diagrams. In *Proc. of IEEE/ACM International Conference on Computer-Aided Design (ICCAD-88)*, pp. 2–5, November 1988.

[FMK91] M. Fujita, Y. Matsunaga, and T. Kakuda. On variable ordering of binary decision diagrams for the application of multi-level logic synthesis. In *Proc. of IEEE The European Conference on Design Automation (EDAC'91)*, pp. 50–54, February 1991.

[FS87] S. J. Friedman and K. J. Supowit. Finding the optimal variable ordering for binary decision diagrams. In *Proc. of 24th ACM/IEEE Design Automation Conference (DAC'87)*, pp. 348–356, June 1987.

[HCO74] S. J. Hong, R. G. Cain, and D. L. Ostapko. MINI: A heuristic approach for logic minimization. *IBM Journal of Research and Development*, Vol. 18, No. 5, pp. 443–458, 1974.

[HMPS94] G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Probabilistic analysis of large finite state machines. In *Proc. of 31st ACM/IEEE Design Automation Conference (DAC'94)*, pp. 270–275, June 1994.

[Ish92] N. Ishiura. Synthesis of multi-level logic circuits from binary decision diagrams. In *Proc. of Synthesis and Simulation Meeting and International Interchange (SASIMI'92, Japan)*, pp. 74–83, March 1992.

[ISY91] N. Ishiura, H. Sawada, and S. Yajima. Minimization of binary decision diagrams based on exchanges of variables. In *Proc. of IEEE/ACM International Conference on Computer-Aided Design (ICCAD-91)*, pp. 472–475, November 1991.

[IY90] N. Ishiura and S. Yajima. A class of logic functions expressible by a polynomial-size binary decision diagrams. In *Proc. of Synthesis and Simulation Meeting and International Interchange (SASIMI'90, Japan)*, pp. 48–54, October 1990.

[IYY87] N. Ishiura, H. Yasuura, and S. Yajima. High-speed logic simulation on vector processors. *IEEE Trans. on Computer-Aided Design*, Vol. CAD-6, No. 3, pp. 305–321, May 1987.

[JT92] R. P. Jacobi and A. M. Trullemans. Generating prime and irredundant covers for binary decision diagrams. In *Proc. of IEEE The European Conference on Design Automation (EDAC'92)*, pp. 104–108, March 1992.

[LCM89] P. Lammens, L. J. Claesen, and H. D. Man. Tautology checking benchmarks: Results with TC. In *Proc. of IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, pp. 600–604, November 1989.

[LL92] 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.

[LPV93] Y.-T. Lai, M. Pedram, and S. B. Vrudhula. FGILP: An integer linear program solver based on function graphs. In *Proc. of IEEE/ACM International Conference on Computer-Aided Design (ICCAD-93)*, pp. 685–689, November 1993.

[LS90] 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.

[MB88] J. C. Madre and J. P. Billon. Proving circuit correctness using formal comparison between expected and extracted behaviour. In *Proc. of 25th ACM/IEEE Design Automation Conference (DAC'88)*, pp. 205–210, June 1988.

[MF89] Y. Matsunaga and M. Fujita. Multi-level logic optimization using binary decision diagrams. In *Proc. of IEEE/ACM International Conference on Computer-Aided Design (ICCAD-89)*, pp. 556–559, November 1989.

[Min92a] S. Minato. Fast generation of irredundant sum-of-products forms from binary decision diagrams. In *Proc. of Synthesis and Simulation Meeting and International Interchange (SASIMI'92, Japan)*, pp. 64–73, April 1992.

[Min92b] S. Minato. Minimum-width method of variable ordering for binary decision diagrams. *IEICE Trans. Fundamentals*, Vol. E75-A, No. 3, pp. 392–399, March 1992.

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

[Min93b] S. Minato. Fast generation of prime-irredundant covers from binary decision diagrams. *IEICE Trans. Fundamentals*, Vol. E76-A, No. 6, pp. 967–973, June 1993.

[Min93c] S. Minato. Fast weak-division method for implicit cube representation. In *Proc. of Synthesis and Simulation Meeting and International Interchange (SASIMI'93, Japan)*, pp. 423–432, October 1993.

[Min93d] 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.

[Min94] S. Minato. Calculation of unate cube set algebra using zero-suppressed BDDs. In *Proc. of 31st ACM/IEEE Design Automation Conference (DAC'94)*, pp. 420–424, June 1994.

[Min95] S. Minato. Implicit manipulation of polynomials using zero-suppressed BDDs. In *Proc. of IEEE The European Design and Test Conference (ED&TC'95)*, pp. 449–454, March 1995.

[MIY89] S. Minato, N. Ishiura, and S. Yajima. Symbolic simulation using shared binary decision diagram. In *Record of the 1989 IEICE Fall Conference, (in Japanese)*, pp. 1.206–1.207, SA-7-5, September 1989.

[MIY90] S. Minato, N. Ishiura, and S. Yajima. Shared binary decision diagram with attributed edges for efficient Boolean function manipulation. In *Proc. of 27th ACM/IEEE Design Automation Conference (DAC'90)*, pp. 52–57, June 1990.

[Miy92] T. Miyazaki. Boolean-based formulation for data path synthesis. In *Proc. of IEEE Asia-Pacific Conference on Circuits and Systems (APCCAS'92)*, pp. 201–205, December 1992.

[MKLC87] 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, June 1987.

[Mor70] E. Morreale. Recursive operators for prime implicant and irredundant normal form determination. *IEEE Trans. on Computers*, Vol. C-19, No. 6, pp. 504–509, June 1970.

[MSB93] P. McGeer, J. Sanghavi, and R. K. Brayton. Espresso-signature: A new exact minimizer for logic functions. In *Proc. of 30th ACM/IEEE Design Automation Conference (DAC'93)*, pp. 618–624, June 1993.

[MWBSV88] S. Malik, A. R. Wang, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Logic verification using binary decision diagrams in a logic synthesis environment. In *Proc. of IEEE/ACM International Conference on Computer-Aided Design (ICCAD-88)*, pp. 6–9, November 1988.

[Oku94] 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.

[Rud93] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proc. of IEEE/ACM International Conference on Computer-Aided Design (ICCAD-93)*, pp. 42–47, November 1993.

[Tak93] A. Takahara. A timing analysis method for logic circuits. In *Record of the 1993 IEICE Spring Conference (in Japanese)*, pp. 1.120, A–20, March 1993.

[THY93] S. Tani, K. Hamaguchi, and S. Yajima. The complexity of the optimal variable ordering of a shared binary decision diagram. Technical Report 93-6, Department of Information Science, Faculty of Science, University of Tokyo, December 1993.

[TIY91] N. Takahashi, N. Ishiura, and S. Yajima. Fault simulation for multiple faults using BDD representation of fault sets. In *Proc. of IEEE/ACM International Conference on Computer-Aided Design (ICCAD-91)*, pp. 550–553, November 1991.

INDEX