The Journal of Formal Methods in System Design, ??, 1–24 (1996)

Arithmetic Boolean Expression Manipulator Using BDDs

SHIN-ICHI MINATO minato@aecl.ntt.jp
NTT LSI Laboratories, 3-1 Morinosato-Wakamiya, Atsugi-shi, Kanagawa Pref., 243-01 Japan.

Received ??, ??, 1995; Revised ??, ??, 1996

Editor:

Abstract. Recently, there has been a lot of works on LSI design systems using Binary Decision Diagrams (BDDs), which are efficient representations of Boolean functions. We previously developed a Boolean expression manipulator, that can quickly calculate Boolean expressions by using BDD techniques. It has greatly assisted us in developing VLSI design systems and solving combinatorial problems.

In this paper, we present an *Arithmetic Boolean Expression Manipulator* (BEM-II), that is also based on BDD techniques. BEM-II calculates Boolean expressions that contain arithmetic operations, such as addition, subtraction, multiplication and comparison, and then displays the results in various formats. It can solve problems represented by a set of equalities and inequalities, which are dealt with in 0-1 linear programming. We discuss the algorithm and data structure used for manipulating arithmetic Boolean expressions and show the formats used for displaying the results.

The specifications for BEM-II are described and several application examples are presented. Arithmetic Boolean expressions will be useful for various applications. They perform well in terms of the total time for programming and execution. We expect BEM-II to facilitate research and development of digital systems.

Keywords: Boolean function, BDD, Boolean expression, LSI CAD, combinatorial problem

Arithmetic Boolean Expression Manipulator Using BDDs

1. Introduction

Manipulation of Boolean functions is an important technique for implementing VLSI design systems and many other problems in computer science. Binary Decision Diagrams (BDDs), which were proposed by Akers[1] and Bryant[2], are graph representations of Boolean functions. BDDs are attracting attention because they enable us to manipulate Boolean functions efficiently in terms of time and space. Algorithms based on conventional data structures, such as truth tables and cube sets, can often be remarkably improved by using BDDs[3][4]. BDD techniques can also be used for solving general covering problems[6][7].

In the research and development of digital systems, Boolean expressions are sometimes used to handle problems and procedures. It is a cumbersome job to calculate

Boolean expressions by hand, even if 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 them 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)[8], which is an interpreter that uses BDDs to calculate Boolean expressions. It enables us to check the equivalence and implications of Boolean expressions quite easily. It has helped us in developing VLSI design systems and solving combinatorial problems.

Although most discrete problems can be described by Boolean expressions, arithmetic operators, such as addition, subtraction, multiplication and comparison, are convenient for describing many practical problems, as seen in 0-1 linear programming. Such expressions can be rewritten using logic operators only, but this can make them complicated and hard to read. In many cases, arithmetic operators provide simple problem descriptions of problems.

In this paper, we present a new Boolean Expression Manipulator, which we call BEM-II, that allows the use of arithmetic operators. 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 efficiently. We developed several output formats for displaying expressions containing arithmetic operators.

The remainder of this paper is organized as follows. In Section 2, we explain the BDD techniques for manipulating ordinary Boolean functions. In Section 3, we explain our method for manipulating Boolean expressions that contains arithmetic operators. In Section 4, we present the implementation of BEM-II and some applications.

2. Boolean Expression Manipulation Using BDDs

A Binary Decision Diagram (BDD) is a directed graph representation of a Boolean function (Fig. 1). BDDs have two terminal nodes, which we call the *0-terminal node* and *1-terminal node*, and many decision nodes with two edges, called the *0-edge* and *1-edge*. A BDD is derived by reducing a binary tree graph, which represents the recursive execution of *Shannon's expansion*.

The following reduction rules give a *Reduced Ordered BDD (ROBDD)*, which represents a Boolean function more efficiently (see [2] for details).

- Eliminate all redundant nodes which have two edges pointing to the same node.
- Share all equivalent sub-graphs.

ROBDDs provide canonical forms for Boolean functions when the variable order is fixed. Most work on BDDs have been based on those reduction rules[9][10]. In the following sections, for the sake of simplification, we refer to ROBDDs as BDDs.

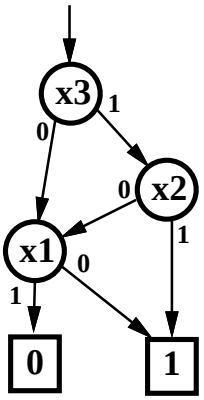
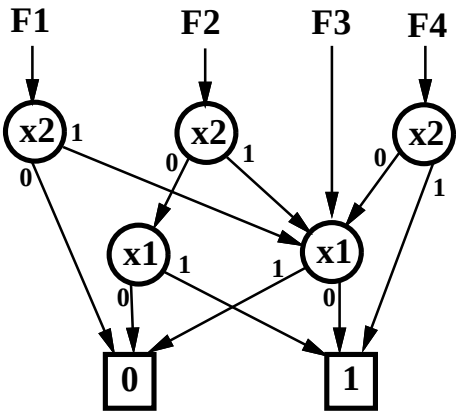


Figure 1. A BDD for $(x3 \cdot x2 \vee x1)$.



$(F1 = x2 \cdot \overline{x1},\ F2 = x2 \oplus x1,\ F3 = \overline{x1},\ F4 = x2 \vee \overline{x1}.\ )$

Figure 2. A shared BDD.

A set of BDDs representing multiple functions can be united into a graph consisting of BDDs sharing their sub-graphs with each other. Manipulation efficiency is improved by managing all the BDDs as a single graph (Fig. 2). We call such graphs *SBDDs (Shared BDDs)*[11]. We can further reduce the operation time and memory requirements by using *attributed edges*[11], which represent such logic operations as inversion.

BDD packages using these techniques exhibit the following useful properties.

- They can generate BDDs for large-scale functions, some of which can not be represented by previous methods.
- After generating BDDs, the equivalence of two functions can be checked in a constant time.
- The time needed for logic operations is almost proportional to the graph size.

Using a BDD package, we can generate BDDs for Boolean functions specified by Boolean expressions. Boolean expressions may consist of a number of input variables and logic operators, such as AND, OR, NOT, and EXOR. A Boolean function can be described with multiple expressions using intermediate variables unless there are cyclic references. To generate BDDs for the expressions, we first define the input variable order and create an BDD that has a single node for each input variable. We then construct complicated BDDs by applying logic operations to the initial BDDs according to the structure of the Boolean expressions.

The computation time for generating the BDDs depends on the length of the Boolean expressions and the size of the BDDs to be generated. It is difficult to estimate the time exactly. We know that the time for one logic operation is approximately proportional to the size of the BDDs. In many cases, the BDDs grow larger with repeated logic operations, unless the expression is redundant. Therefore, the final few logic operations take the most time, and roughly speaking, the total computation time is approximately proportional to the size of the final BDDs.

BDD size 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 can find an adequate order in many cases[11][12][13][15].

After generating BDDs for Boolean expressions, we can use them to:

- check for tautologies of the expressions,
- check for equivalence or implication between pairs of expressions,
- find a counterexample when the above checking fails.
- simplify complicated expressions,
- search for a solution (satisfiable input vector) to the expressions,
- enumerate the possible solutions to the expressions, and
- evaluate the complexity of the expressions.

To utilize these capabilities, we previously developed a Boolean Expression Manipulator (BEM)[8]. This program is an interpreter with a lexical and syntax parser for calculating Boolean expressions. It has helped us develop digital systems and solve combinatorial problems. This program provides several formats for displaying Boolean functions represented by BDDs, as follows.

Karnaugh map Karnaugh map representation is a good way to observe function features. It is practical, however, only for less than five or six inputs. If there are

more inputs but any of them are irrelevant to the function, we can reduce the map by using only the relevant variables. This reduction can easily be done by using BDD operations.

Sum-of-products format The sum-of-products format (also called the PLA format, cube set, or two-level logic) is another good way to display Boolean functions since it clearly shows solutions that satisfy the function. Using the fast generation method presented in [14], we can quickly generate an irredundant sum-of-products (ISOP) format from BDDs. This format is suitable for tautology checking and displaying counterexamples.

BDD representation Several kinds of Boolean functions, such as parity functions, require an exponential-length sum-of-products format, while they can be compactly represented in BDDs. In such cases, the functions can be described by using multiple Boolean expressions with intermediate variables for the respective BDD nodes. If a graphic utility is available, a schematic display of the BDDs using circles and arrows will facilitate understanding.

Statistical information When the function is too complex to display all at once, it is useful to output the statistical information, such as the number of solutions, the density of the truth table (ratio of 0/1), the number of BDD nodes, the ISOP format length, and the number of relevant input variables. This information can be computed efficiently using BDD operations.

Satisfiable solutions We do not have to display Boolean functions completely when we seek solutions or counterexamples to a problem. In many cases, any one solution can be easily shown, even if the function is too complicated to display all at once. By traversing the BDDs, we can find one of the solutions in a time proportional to the number of inputs.

When each input variable has a cost, the minimum-cost solution can be computed from the BDDs[6]. Namely, where

$$Cost = \sum_{i=1}^{n} w_i \cdot x_i \quad (w_i > 0,\ x_i \in \{0, 1\}),$$

we can find a solution for $x_1, x_2, \ldots, x_n$ which minimizes $Cost$. Searching for the minimum-cost solution is done by backtracking through the BDDs. Using a cache-based technique, we can find the minimum-cost solution in a time proportional to the number of nodes in the BDDs.

This method enables us to solve various problems automatically by describing them using Boolean expressions. Many NP complete problems can be solved immediately if the BDDs for them can be generated in the main memory of the computer. Of course, they are still problems in NP, so in general, the BDDs require an exponential number of nodes and thus overflow the memory. However, there are many practical examples where the BDDs become surprisingly compact. They are therefore useful for researching digital systems and implementing prototype programs.

3. Manipulation of Arithmetic Boolean Expressions

As we discussed above, although most discrete problems can be described by using Boolean expressions, 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

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

Using only Boolean expressions, this function become 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 that uses BDDs to represent and manipulate expressions with arithmetic operators.

3.1. Definitions

For manipulating Boolean expressions that include arithmetic operators, we define *arithmetic Boolean expressions* and *Boolean-to-integer functions*, which are extended models of conventional Boolean expressions and Boolean functions.

Arithmetic Boolean expressions are extended Boolean expressions which contain not only logic operators, but also arithmetic operators, such as addition ($+$), subtraction ($-$), and multiplication ($\times$). Any integer number is allowed to be used as a constant term in the expression, but input variables are restricted to either 0 or 1. Equality ($=$) and inequalities ($<, >, \leq, \geq, \neq$) are defined as operations which return 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 another example.

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 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 $x_1 = x_2 = 0$, or 4 when $x_1 = x_2 = 1$. We can see that an arithmetic Boolean expression represents a function from binary-vector to integer: $(B^n \rightarrow I)$. We call this function a **Boolean-to-integer (B-to-I) function**. The operators in arithmetic Boolean expressions are defined as operations on B-to-I functions. We can calculate B-to-I functions for arithmetic Boolean expressions by applying operations on B-to-I functions according to the structure of the expressions.

	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 3. Computation of arithmetic Boolean expressions.

The procedure for obtaining the B-to-I function for the expression $(3 \times x_1 + x_2 < 4)$ is shown in Fig. 3. First, multiply the constant function 3 times 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 $(\overline{x_1} \vee \overline{x_2})$.

3.2. Representation of B-to-I Functions

Figure 3 showed how a B-to-I function can be obtained by enumerating the output values for all possible combinations of the input values. This is impracticable when there are many input variables since the number of combinations grows exponentially. We thus need a more efficient way to represent B-to-I functions.

There are two ways to represent B-to-I functions using BDDs: *Multi-Terminal BDDs (MTBDDs)* and *BDD vectors.*

MTBDDs are the extended BDDs with multiple terminal nodes, each of which has an integer value (Fig. 4). This method is natural and easy to understand; however, we need to develop a new BDD package to manipulate multi-terminals. Hachtel and Somenzi et al. have reported several works[16, 17] on MTBDDs. They call MTBDD in other words, *Algebraic Decision Diagrams (ADDs)*,

BDD vectors is the way to represent B-to-I functions with a number of usual BDDs. By encoding the integer numbers into n-bit binary codes, a B-to-I function can be decomposed into n pieces of Boolean functions that represent the respective bits as either 1 or 0. These Boolean functions can then be represented with BDDs which are shared each other (Fig. 5). This method was mentioned in [18].

Here we discuss which representation is more efficient in terms of size. We show two typical examples that are in contrast to each other.

1. Assume a multi-terminal BDD with a large number of decision nodes and n terminals with random values of n-bit integers (Fig. 6(a)). If we represent the same function by using an n-bit BDD vectors, these BDDs can hardly share their sub-graphs; therefore, the BDD vector requires about n times the number of nodes as the multi-terminal BDD (Fig. 6(b)).

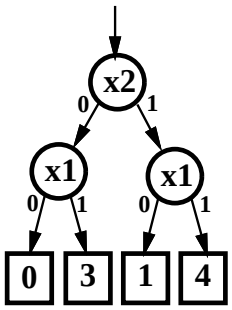


Figure 4. A multi-terminal BDD (MTBDD).

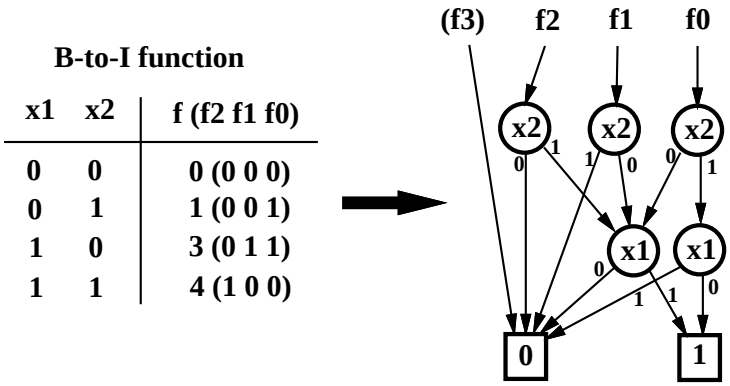
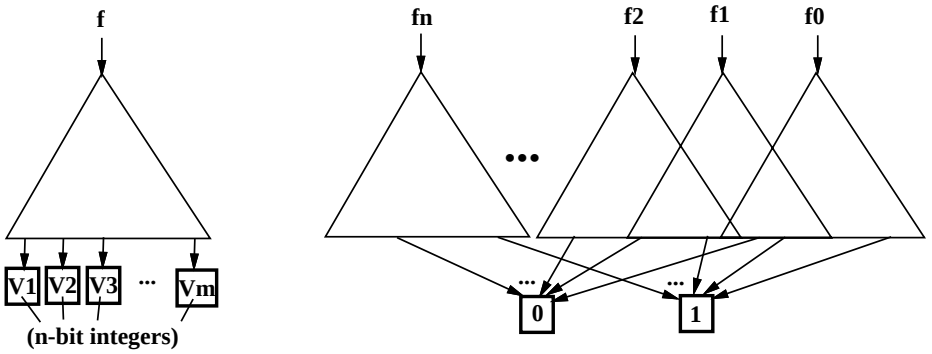


Figure 5. A BDD vector.

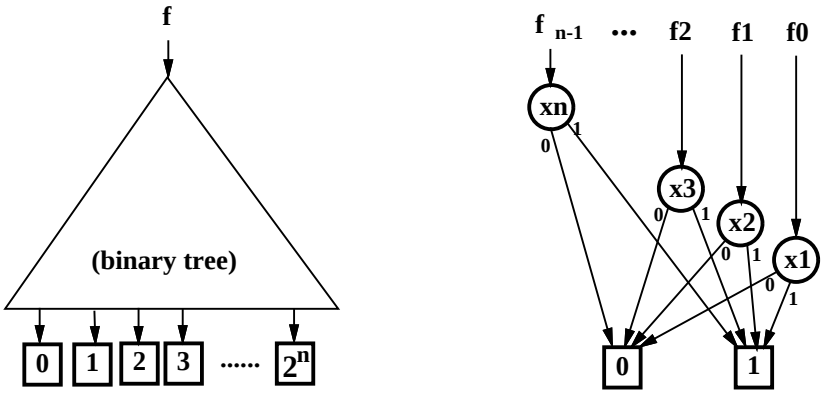
2. Assume a B-to-I function for $(x_1 + 2 \times x_2 + 4 \times x_3 + \ldots + 2^{n-1} \times x_n)$. This function can be represented by an n-nodes BDD vector. However, we need 2^n terminals to use a multi-terminal BDD (Fig. 7).

We show that the comparison between multi-terminal BDDs and BDD vectors can be reduced to the variable-ordering problem. Assume the BDD shown in Fig. 8(a), which was obtained by combining the BDD vector shown in Fig. 5 with what we call *bit-selection variables*. If we change the variable order to move the bit-selection variables from higher to lower position, the BDD becomes as shown in Fig. 8(b). In this BDD, the sub-graphs with bit-selection variables correspond to the terminals in the multi-terminal BDD. Namely, multi-terminal BDDs and BDD vectors can be transformed into each other by changing the variable order, assuming bit-selection variables. This observation indicates that the efficiency of the two representations



(a) MTBDD. (b) BDD vector.

Figure 6. An example where MTBDD is better.



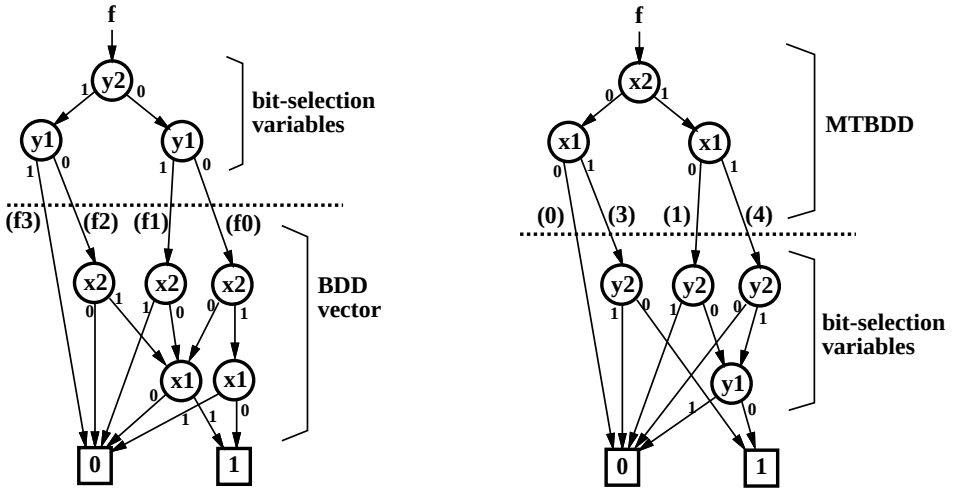
(a) MTBDD. (b) BDD vector.

Figure 7. An example where BDD vector is better.

depends on the nature of the objective functions. If we know the functions to be generated, the appropriate location of bit-selection variables will be found.

We then considered which representations are more favorable for implementing arithmetic Boolean expression manipulator. In this application, 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). However, multi-terminal BDDs cannot be shared (Fig. 10). We therefore 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 under



(a) BDD vector with bit-sel. variables. (b) MTBDD with bit-sel. variables.

Figure 8. Bit-selection variables.

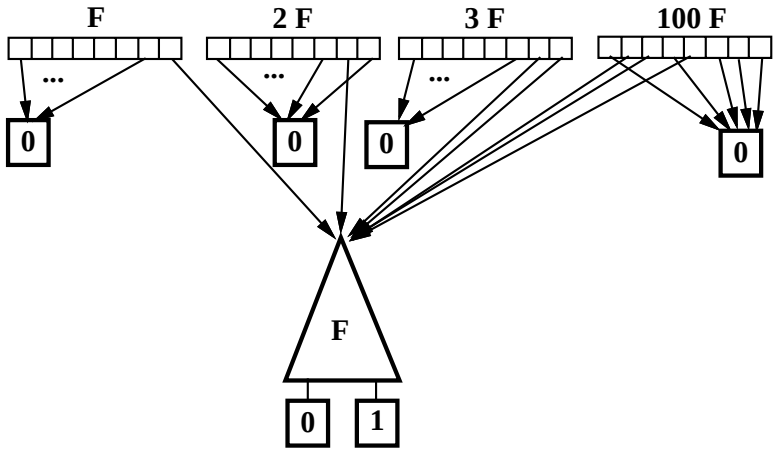


Figure 9. BDD vectors for arithmetic Boolean expressions.

which conditions the B-to-I function produces a negative value. This coding scheme requires specifying the word-length to know which is sign bit. An easy way is to allocate a long length in advance, but it limits the range of numbers. In our implementation, we supported variable word-length for each data, so there is no limit on the range of numbers.

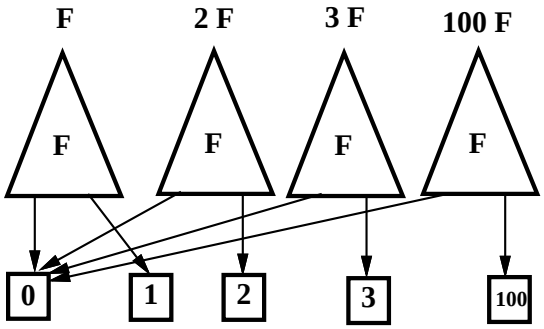


Figure 10. MTBDDs for arithmetic Boolean expressions.

3.3. 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, a new B-to-I function is generated. We define two kinds of inversion operations: bit-wise inversion and logical inversion. Logical inversion returns 1 only for 0, otherwise it returns 0.

Arithmetic addition can be composed using logic operations on BDDs by simulating a conventional hardware algorithm of full-adders which are 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, such as 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 indicated by 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 is useful if we can 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 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 with only 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 the same 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:

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

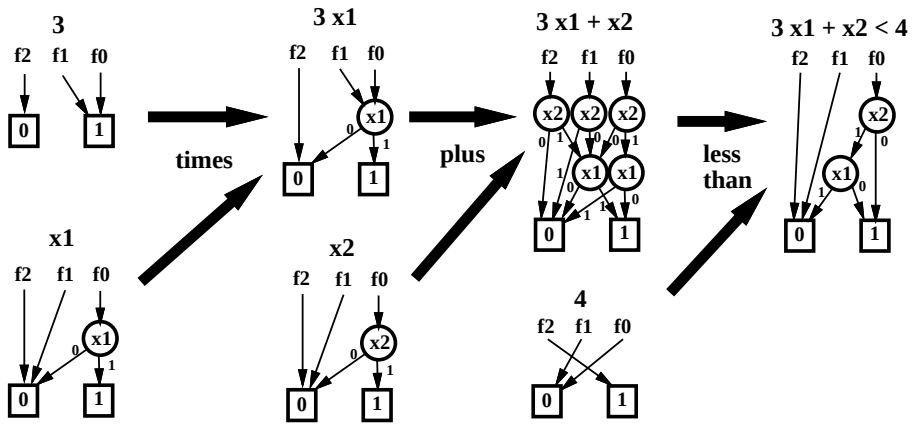


Figure 11. Generation of BDD vectors for arithmetic Boolean expressions.

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

An example of calculating arithmetic Boolean expressions using BDD vectors is shown in Fig. 11.

3.4. 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 using a matrix of logic values (0, 1). We extended the Karnaugh map to use integer numbers for each element (Fig. 12). We call this an *integer Karnaugh map*. It is useful for observing the behavior of B-to-I functions. Like ordinary Karnaugh maps, they are practical only for fewer than five or six input functions. 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.

Bit-wise 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 16 shows the bit-wise expression for the same function shown in Fig. 13.

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 12. An integer Karnaugh map.

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

$$\begin{aligned}
\pm &: (\overline{a} \wedge c \wedge \overline{d}) \vee (\overline{b} \wedge c) \\
f_2 &: (a \wedge b \wedge \overline{c}) \vee (\overline{a} \wedge c \wedge \overline{d}) \vee (b \wedge \overline{c} \wedge d) \vee (\overline{b} \wedge c) \\
f_1 &: (a \wedge \overline{b}) \vee (a \wedge d) \vee (\overline{a} \wedge b \wedge \overline{d}) \\
f_0 &: (b \wedge \overline{d}) \vee (\overline{b} \wedge d)
\end{aligned}$$

Figure 13. A bit-wise 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 14. Case enumeration format.

Bit-wise expression is not so 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.

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 bit-wise 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. 14). 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 $(v_1 \times f_1 + v_2 \times f_2 + \ldots + v_n \times f_m)$.

Using this method, $(2 \times a + 3 \times b - 4 \times c + d)$ would be displayed as

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

This expression seems too complicated compared to 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 of arithmetic addition, subtraction, and multiplication operators only. 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 sub-functions 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 if it is possible. For example, the B-to-I function for $2 \times (a + 3 \times b) - 4 \times (a + b)$ 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, $(a \wedge \overline{b}) \vee (c \wedge \overline{d})$ becomes $-a\ b\ c\ d + a\ b\ c - a\ b + a\ c\ d - a\ c + a - c\ d + c$. This expression is more difficult to read than the original one.

4. Applications of Arithmetic Boolean Expression Manipulator

Using the techniques described above, we developed an arithmetic Boolean expression manipulator. This program, called BEM-II, is an interpreter with a lexical

Table 1. Operators in BEM-II (The upper operators are executed prior to 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( )`

and syntax parser for calculating arithmetic Boolean expressions and displaying the results in various formats. This section gives the specifications for BEM-II and discusses some applications.

4.1. BEM-II Specification

BEM-II 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 list of available operators is shown in Table 1, and an execution example is shown in Fig. 15.

In BEM-II scripts, we can use two kind of variables, *input variables* and *register variables*. Input variables, denoted by strings starting with a lower-case 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 upper-case letter, are used to identify the memory to which a B-to-I function 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, not using register variables. BEM-II allows 65,535 different input variables to be used. There is no limit on the number of register variables.

BEM-II supports such logical operators such as AND, OR, EXOR, and NOT, and such arithmetic operators as plus, minus, multiply, division, shift, equality, inequality, and upper/lower bound. The syntax of expressions generally conforms to C language specifications. The expression `A : B ? C` means *if-then-else*, equivalent

```
% 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
 +-: !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 15. An example of executing BEM-II.

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 is enough to calculate expressions that used to be manipulated by hand, of course. The results can be displayed in the various formats presented in earlier sections.

The input variables are assumed to have 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 having the integer value from 0 to 15. In another way, `X = x1 + 2*x2 + 3*x3 + 4*x4` represents a variable having 1 to 4, under the one-hot constraint `(x1 + x2 + x3 + x4 == 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 compose scripts for various applications much more easily than developing and tuning a specific program.

Here we show the application of BEM-II to several practical problems.

4.2. Subset-Sum Problem

BEM-II can be utilized for many combinatorial problems. *Subset-sum problem* is one example of a problem that can easily be described by arithmetic Boolean expressions and be solved by BEM-II. This problem is to find a subset whose total is equal to a given number b, chosen from a given set of positive integers $\{a_1, a_2, a_2, \ldots, a_n\}$. It is a basic and important problem for many applications including VLSI CAD systems.

In BEM-II script, we use n input variables for representing whether the i-th number is chosen or not. The constraint of these input variables can be described with simple arithmetic Boolean expressions. The followings 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)
```

`C` means the condition of the input variables to satisfy the constraint. This expression is almost the same as the definition of the problem. We can easily read and write the script.

BEM-II is convenient not only to solve the problem but also to analyze the nature of the problem. We can analyze the behavior of the constraint functions in various format. An example of execution is shown in Fig. 16.

```
% 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 16. Execution of BEM-II for a subset-sum problem.

4.3. 8-Queens Problems

8-Queens problem is another example of BEM-II application. BDD-based computation is not so remarkably effective in this problem, but it is a good example to show that BEM-II is very convenient to describe the problem.

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.

Table 2. Results for N-queens problems.

N	#Variable	#BDD	#Solution	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

- 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 )
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 2 shows the results when we applied this method to the 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 a larger N by using a conventional algorithm based on backtracking. However, the conventional method does not enumerate all the solutions nor 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 quite 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.

4.4. Traveling Salesman Problem

Traveling salesman problem (TSP) can also be solved by using BEM-II. The problem is finding the minimum cost path to visit all the cities once and return the start city.

Assume 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, the constraints of the TSP can be expressed as follows.

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

Table 3. Results for TSP.

n	#solutions	BDD size	time(s)
8	2520	2054	8.7
9	66136	20160	28.8
10	181440	19972	216.5

...
$x_{1n} + x_{2n} + \ldots + x_{(n-1)n} = 2$

- All the cities are connected:

  step1
  Let $F_1 = 1, \quad F_2, F_3, \ldots, F_n = 0$.

  Repeat step2 for n times.

  step2
  Let $F_1 = x_{12}F_2 \vee x_{13}F_3 \vee \ldots \vee x_{1n}F_n$.
  Let $F_2 = x_{12}F_1 \vee x_{23}F_3 \vee \ldots \vee x_{2n}F_n$.
  ...
  Let $F_n = x_{1n}F_1 \vee x_{2n}F_2 \vee \ldots \vee x_{(n-1)n}F_{n-1}$.

  step3 Condition $C = F_1 \wedge F_2 \wedge \ldots \wedge F_n$.

The logical product of all the above constraint expressions becomes the solution to the TSP. BEM-II feeds this 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 shown in Table 3. We can solve the problem up to $n = 10$. This seems poor since conventional method solves more than $n = 1000$. However, our method computes all the solutions at once, and the additional constraints can be specified flexibly. For example,

- There is a path which should be used, or not be used.
- There is a city which sould be visited first (second, third, ...).
- The start city and goal 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.

4.5. Timing Analysis for Logic Circuits

For designing high-speed digital systems, timing analysis of logic circuits is important. The orthodox approach is to traverse the circuit to find the active path with

Table 4. Results of timing analysis.

Circuit	In	Out	Gate	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

the topologically maximum length. Takahara[19] proposed a new timing analysis method using 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 on each gate. For example, on a two-input AND gate with delay D, where 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 an arithmetic Boolean expression as

```
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. Table 4[19] shows the experimental results for practical benchmark circuits. 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. Utilizing BEM-II, we can then analyze the circuits in various ways. For example, we can easily compare the delay times between two nets in the circuit.

4.6. Scheduling Problem in Data Path Synthesis

Scheduling is one of the most important subtasks that must be solved to perform data path synthesis. Miyazaki[20] proposed a method for solving scheduling problems 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

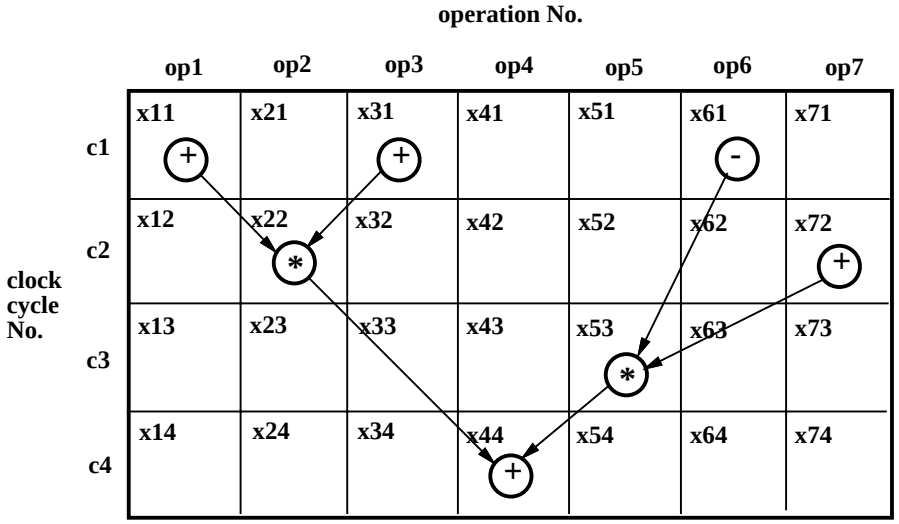


Figure 17. An example of data-flow graph.

operation units and the maximum clock cycles (Fig. 17). While this scheduling problem can be solved by using linear programming, BEM-II can also be utilized.

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

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

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 $C_1 = 1 \times x_{11} + 2 \times x_{12} + \ldots + n \times x_{1n}$.

Table 5. Results of scheduling problem.

data	#all solutions	#optimal solusions	c-step	#multi-plier	#ALU	BDD size	time(s)
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

Let $C_2 = 1 \times x_{21} + 2 \times x_{22} + \ldots + n \times x_{2n}$.
...
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 the above 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. Otherwise it may abort. Table 5[20] shows 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.

5. 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. BEM-II can directly analyze 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 to solve the problem in a programming language. Besides the examples we have shown, BEM-II can also be utilized for minimum-tree problems, magic squares, crypt-arithmetic problems, etc. Although the computation speed is second to 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 for researching and developing digital systems.

Acknowledgments

The author wish to acknowledge the interesting discussions with Atsushi Takahara, Toshiaki Miyazaki, Hiroshi Okuno, and Masayuki Yanagiya.

References

1. S. Akers: Binary Decision Diagrams, IEEE Trans. Comput., Vol. C-27, No. 6, pp. 509-516, June 1978.
2. R. Bryant: Graph-Based Algorithms for Boolean Function Manipulation, IEEE Trans. Comput., Vol. C-35, No. 8, pp. 677-691, Aug. 1986.
3. S. Minato, N. Ishiura and S. Yajima: 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, Nov. 1989.
4. Y. Matsunaga and M. Fujita: Multi-level Logic Optimization Using Binary Decision Diagrams, Proc. ICCAD'89, pp. 556-559, Nov. 1989.
5. J. Burch, E. Clarke, K. McMillan and D. Dill: Sequential Circuit Verification Using Symbolic Model Checking, Proc. ACM/IEEE DAC'90, pp.618-624, June 1992.
6. B. Lin and F. Somenzi: Minimization of Symbolic Relations, Proc. IEEE ICCAD'90, pp. 88-91, Nov. 1990.
7. S. Jeong and F. Somenzi: A New Algorithm for the Binate Covering Problem and its Application to the Minimization of Boolean Relations, Proc. IEEE ICCAD'92, pp. 417-420, Nov. 1992.
8. S. Minato, N. Ishiura and S. Yajima: "Symbolic Simulation Using Shared Binary Decision Diagram", Record of 1989 IEICE National Convention, SA-7-5, pp.1.206-207, (in Japanese) Sep. 1989.
9. K. Brace, R. Rudell and R. Bryant: "Efficient Implementation of a BDD Package", ACM/IEEE Proc. 27th DAC, pp. 40-45, June 1990.
10. R. Bryant: "Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams", CMU CS technical report, No. CMU-CS-92-160, July 1992.
11. S. Minato, N. Ishiura and S. Yajima: Shared Binary Decision Diagram with Attributed Edges for Efficient Boolean Function Manipulation, ACM/IEEE Proc. 27th DAC, pp. 52-57, June 1990.
12. M. Fujita, Y. Matsunaga and T. Kakuda: 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
13. S. Minato: Minimum-Width Method of Variable Ordering for Binary Decision Diagrams, IEICE Jpn. Trans. Fundamentals, Vol. E75-A, No. 3, pp. 392-399, Mar. 1992.
14. S. Minato: 'Fast Generation of Irredundant Sum-of-Products Forms from Binary Decision Diagrams', Proceedings of the Synthesis and Simulation Meeting and International Interchange (SASIMI'92, Japan), pp. 64-73, Mar. 1992.
15. R. Rudell: "Dynamic Variable Ordering for Ordered Binary Decision Diagrams", Proc. IEEE/ACM ICCAD'93, pp. 42-47, Nov. 1993.
16. I. Bahar, E. Frohm, C. Gaona, G. Hachtel, W. Macil, A. Pardo and F. Somenzi: "Algebraic Decision Diagrams and Their Applications", Proc. IEEE/ACM ICCAD'93, pp. 188-191, Nov. 1993.
17. G. Hachtel, E. Macii, A. Pardo, and F. Somenzi: "Probabilistic Analysis of Large Finite State Machines", Proc. ACM/IEEE DAC'94, pp. 270-275, June 1994.
18. E. Clarke, K. McMillan, X. Zhao, M. Fujita and J. Yang: Specttral Transforms for Large Boolean Functions with Applications to Technology Mapping, Proc. ACM/IEEE DAC'93, pp. 54-60, June 1993.
19. A. Takahara: "A Timing Analysis Method for Logic Circuits", Record of 1993 IEICE National Convention, A-120, p. 1-120, (in Japanese), Mar. 1993.
20. T. Miyazaki: "Boolean-Based Formulation for Data Path Synthesis", IEEE Asia-Pacific Conference on Circuits and Systems (APCCAS'92), pp. 201-205, Dec. 1992.

Received Date: ??, ??, ??
Accepted Date: ??, ??, ??
Final Manuscript Date: June 23, 1996.