PAPER Special Section on VLSI Design and CAD Algorithms

BEM-II: An Arithmetic Boolean Expression Manipulator Using BDDs

Shin-ichi MINATO†, *Member*

SUMMARY Recently, there has been a lot of research on solving combinatorial problems using Binary Decision Diagrams (BDDs), which are very efficient representations of Boolean functions. We have already developed a Boolean Expression Manipulator, which calculates and reduces Boolean expressions quickly based on BDD techniques. This greatly aids our works on developing VLSI CAD systems and solving combinatorial problems. Any combinatorial problem can be described in Boolean expressions; however, arithmetic operations, such as addition, subtraction, multiplication, equality and inequality, are also used for describing many practical problems. Arithmetic operations provide simple descriptions of problems in many cases. In this paper, we present an *arithmetic Boolean expression manipulator* (BEM-II), based on BDD techniques. BEM-II calculates Boolean expressions containing arithmetic operations and then displays the results in various formats. It can solve problems represented by a set of equalities and inequalities, which are dealt with using 0-1 linear programming. We show the efficient data structure based on BDD representation, algorithms for manipulating Boolean expressions with arithmetic operations, and good formats for displaying the results. Finally we present the specification of BEM-II and an example of application to the 8-Queens problem. BEM-II is customizable to various applications. It has good computation performance in terms of the total time for programming and execution. We expect BEM-II to be a helpful tool in research and development on digital systems.

key words: *BDD (binary decision diagram), Boolean function, arithmetic Boolean expression, B-to-I (Boolean-to-integer) function, combinatorial problem*

1. Introduction

Manipulating Boolean functions is an important technique for implementing VLSI CAD systems and for various problems in the theory of algorithms and complexity. Binary Decision Diagrams (BDDs), which were proposed by Akers [1] and Bryant [2], are graph representations of Boolean functions. Recently, BDDs have attracted much attention because they enable us to manipulate Boolean functions efficiently in terms of time and space. There are many cases where the algorithm based on conventional Boolean function representations, such as truth tables or cube sets, can be improved and efficiently adapted for BDD operations [3], [4]. Besides VLSI CAD systems, BDDs can be used in the method of solving binate covering problems [5], [6], which has various applications.

In researching and developing digital systems, we sometimes describe and calculate Boolean expressions to consider problems or procedures. It is a cumbersome job to calculate or reduce Boolean expressions by hand, even if the expressions have few variables. In cases having more than 5 or 6 variables, we might as well give up.

We have already developed a Boolean Expression Manipulator (BEM) [7]. That program calculates and reduces Boolean expressions quickly based on BDD techniques. Expressions which can hardly be manipulated by hand can be processed in a second using BEM. It enables us to check the equivalence and implication of Boolean expressions easily. It greatly helped our work on developing VLSI CAD systems and solving combinatorial problems.

Any combinatorial problem can be described in Boolean expressions; however, arithmetic operations, such as addition, subtraction, multiplication, equality and inequality, are also used for describing many practical problems, as seen in linear-integer programming. Such expressions can be rewritten using logic operations only, but they would be complex and hard to read. Arithmetic operations provide simple descriptions of problems in many cases.

In this paper, we present a new Boolean expression manipulator BEM-II, which allows the use of arithmetic operations. BEM-II can directly solve problems represented by a set of equalities or inequalities, which are dealt with using 0-1 linear programming. Of course, BEM-II can also manipulate ordinary Boolean expressions as it incorporates the original BEM. Furthermore, the output formats have been improved. We show the data structure, algorithms for calculating Boolean expressions with arithmetic operations, and good formats for displaying results.

The remainder of this paper is organized as follows. In Sect. 2, we explain the techniques for manipulating ordinary Boolean functions using BDDs. In Sect. 3, we show a method for manipulating Boolean expressions with arithmetic operations. In Sect. 4, we present the implementation of BEM-II and its applications.

Manuscript received March 22, 1993.
Manuscript revised May 18, 1993.
† The author is with NTT LSI Laboratories, Atsugi-shi, 243-01 Japan.

2. Boolean Expression Manipulation Using BDDs

2. 1 BDD

A Binary Decision Diagram (*BDD*) is a directed graph representation of a Boolean function, as shown in 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, as shown in Fig. 2. The binary tree represents the recursive execution of *Shannon's expansion*.

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

(1) Eliminate all the redundant nodes whose two edges point to the same node.

(2) Share all the equivalent sub-graphs.

ROBDDs give canonical forms for Boolean functions when the variable order is fixed. Most works on BDDs are based on the above reduction rules. In the following sections, for the sake of simplification, we refer to ROBDDs as BDDs (or original BDDs).

Since there are 2^{2^n} kinds of n-input Boolean functions, the representation requires at least 2^n bit of memory in the worst case. It is known that a BDD for an n-input function includes $O(2^n/n)$ nodes in general [8]. As each node consumes about $O(n)$ bit (to distinguish the two child nodes from $O(2^n/n)$ nodes), the total storage exceeds 2^n bit. However, the size of BDDs varies with the kind of function, unlike truth tables which always require 2^n bit of memory. There is a class of Boolean functions that can be represented by a polynomial size of BDDs, and many practical functions fall into this class [9]. This is an attractive feature of BDDs.

A set of BDDs representing multiple functions can be united into a graph which consists of BDDs sharing their sub-graphs with each other. The efficiency of manipulation can be improved by managing all the BDDs as a single graph, as in Fig. 3. We call such graphs *SBDDs* (*Shared BDDs*) [10]. We can further reduce the operation time and memory requirement by using *attributed edges* [10], which represent certain logic operations such as inversion.

BDD packages implementing such techniques exhibit the following useful properties.

- They can generate BDDs for large-scale functions, some of which have never been represented before by previous methods.
- After generating BDDs, the equivalence of two functions can be checked in a constant time.
- Logic operations can be carried out within a time that is almost proportional to the graph size.

2. 2 Generation of BDDs from Boolean Expressions

Here we show the method of generating BDDs for the functions of given Boolean expressions.

1. Define the order of input variables, such as x_1, x_2, $\cdots$, x_n.
2. Make a BDD with a single node for each input variable.
3. Construct more complex BDDs by applying logic operations on BDDs according to the Boolean expressions.

An example for $(x_1 \wedge x_2) \vee x_3$ is shown in Fig. 4. First, trivial BDDs for x_1, x_2, x_3 are generated. Then applying the AND operation between x_1 and x_2, the BDD of $x_1 \wedge x_2$ is generated. The final BDD for the entire expression is obtained as the result of the OR operation between $x_1 \wedge x_2$ and x_3. In this manner, we can also generate BDDs for functions given as multiple expressions using internal variables.

The computation time for generating BDDs

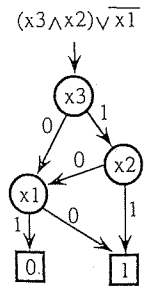


Fig. 1 A BDD.

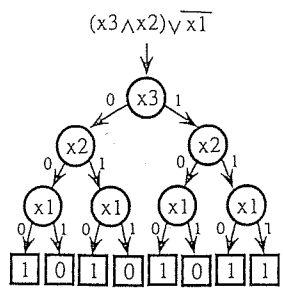


Fig. 2 A binary decision tree.

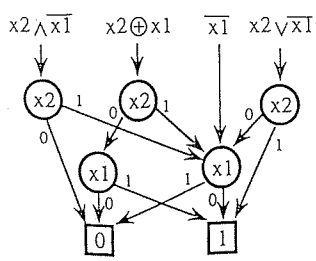


Fig. 3 A shared BDD.

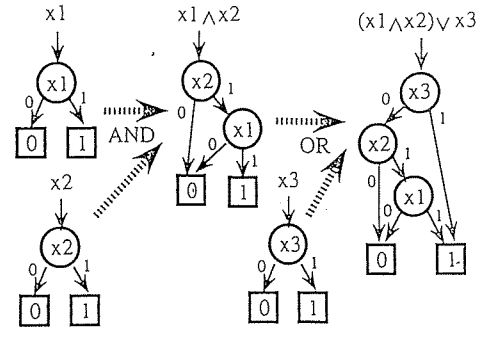


Fig. 4 Generation of BDDs from Boolean expressions.

depends on the length of Boolean expressions and the size of the BDDs to be generated. Up to now, it has been difficult to estimate the time exactly. We know that the time for one logic operation is approximately proportionaly 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 occupy most of the time, and roughly speaking, the total computation time is approximately proportional to the size of the final BDDs.

The size of BDDs largely depends on the order of the input variables. It is difficult to derive a method that always yields the best order, but with some heuristic methods, we are able to find an adequate order in many cases [10]-[12].

2.3 Display Formats of Boolean Functions

After the generation and manipulation of BDDs for Boolean expressions, the results are displayed in a certain format adapted for calculation. We assume the following utilities for aiding research on digital systems.

- Tautology checking of Boolean expressions.
- Equivalence or implication checking between two expressions.
- Finding a counterexample when the above checking failed.
- Simplification of complicated expressions.
- Searching for a solution (satisfiable input) of the Boolean expression.
- Enumerating or counting the solutions of the expression.
- Evaluating the complexity of expressions.

Considering these operations, we provide several formats to display Boolean functions represented by BDDs.

Karnaugh map: Unless the number of input variables is large, a Karnaugh map representation (Fig. 5) is a good way to observe the feature of functions. We can readily check tautology or inconsistency by viewing the map. However, it is practicable only for less than six input functions as the map size grows exponentially. When there are too many inputs but some of them are irrelevant to the function, we can reduce the map by excluding such input variables. The relevance checking can be implemented efficiently using BDD operations.

Sum-of-products format: As shown in Fig. 6, the sum-of-products format (also called PLAs, cube sets, or two-level logic) is another good method of displaying Boolean functions since it enumerates satisfiable solutions of the function. Using the method presented in [13], we can quickly generate an irredundant sum-of-products (ISOP) format from BDDs. ISOP format can be utilized to evaluate a kind of complexity of

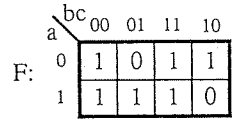


Fig. 5 A Karnaugh map.

$$F=(a\wedge\overline{b})\vee(\overline{a}\wedge\overline{c})\vee(b\wedge c)$$

Fig. 6 An irredundant sum-of-products format.

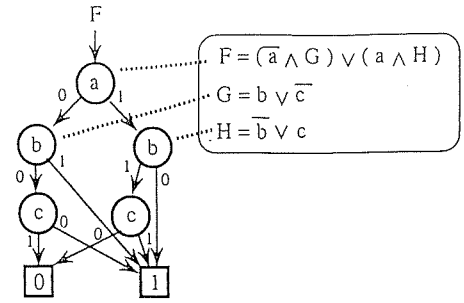


Fig. 7 A multi-level Boolean expression for a BDD.

Boolean functions. Tautology checking can easily be performed by looking at the ISOP format.

BDD representation: Some kinds of Boolean functions, such as parity functions, require exponential length expressions to display them in the sum-of-products format, however, they can be represented in BDDs compactly. In such cases, it is useful to display BDDs graphically. If no graphic utility is available, there is a method of displaying BDDs with multi-level Boolean expressions by assigning an internal variable to each node of the BDDs, as shown in Fig. 7.

Statistical information: When the function is too complex to display all at once, it is useful to output statistical information, such as the number of solutions, density of truth table (ratio of 0/1), number of nodes in the BDDs, length of ISOP format, and number of relevant input variables. These data 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 given as a Boolean expression. In many cases, any one of solutions can be shown quite easily, even if the function is too complex to display. By traversing BDDs, we can find a solution in a time proportional to the number of inputs.

2.4 Application to Combinatorial Problems (1)

We introduce a method of solving combinatorial problems using BDDs [5]. Here we consider the problems that seek a combination of values to inputs which gives the minimum at a *cost function and satisfies a constraint function*. Namely, where *cost function:*

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

constraint function:

$$f(x_1, x_2, \cdots, x_n) \in \{0, 1\},$$

to seek values for $x_1, x_2, \cdots, x_n$ which makes *Cost* minimum under the constraint $f=1$. Many NP complete problems can be described in the above format.

To solve the problem using BDDs, we first generate a BDD for f. In the BDD, the set of paths from the root node to the terminal node with the '1' value corresponds to the solutions of the problem. On each path, the edges labeled '1' represent assigning the value '1' to the input, namely it takes the cost for the input. Therefore, we may find a path to the '1' terminal node in the BDD so that the total cost of '1' edges is minimum.

Searching for the minimum cost path is implemented based on back tracking of the BDD. It appears to take an exponential time, but we can avoid duplicate tracking for shared subgraphs in the BDD by storing the minimum cost for the subgraph and referring to it at the second visit. This technique eliminates the need to visit each node more than once, so we can find the minimum cost path in a time proportional to the number of nodes in the BDD.

In this method, we can immediately solve the problem if the BDD for the constraint function can be generated in the main memory of the computer. Of course, it is still a problem in NP, so in general the BDD requires an exponential number of nodes and overflows the memory. However, there are many practical examples where the BDD becomes surprisingly compact.

The BDD-based method features customizability. We can automatically solve any problem if it is described as Boolean expressions. In terms of computation time and storage, this method may not be as good as conventional methods which are devised by effective heuristics for a specific problem. The BDD-based method appears suitable for implementing prototypes for aiding research on algorithms in digital systems.

3. Manipulation of Boolean Expressions Including Arithmetic Operations

It is possible to describe any combinatorial problem as Boolean expressions; however, arithmetic operations, such as addition, subtraction, multiplication, equality and inequality, are also used for describing many practical problems, as seen in linear-integer programming. For example, a majority function with five inputs can be expressed concisely using arithmetic operations as:

$$x_1+x_2+x_3+x_4+x_5 \geqq 3,$$

otherwise it becomes a difficult expression as:

$$(x_1 \wedge x_2 \wedge x_3) \vee (x_1 \wedge x_2 \wedge x_4) \vee (x_1 \wedge x_2 \wedge x_5)$$
$$\vee (x_1 \wedge x_3 \wedge x_4) \vee (x_1 \wedge x_3 \wedge x_5) \vee (x_1 \wedge x_4 \wedge x_5)$$
$$\vee (x_2 \wedge x_3 \wedge x_4) \vee (x_2 \wedge x_3 \wedge x_5) \vee (x_2 \wedge x_4 \wedge x_5)$$
$$\vee (x_3 \wedge x_4 \wedge x_5).$$

In this section, we show an efficient method of representing and manipulating such expressions including arithmetic operations using BDDs.

3.1 Definitions

For manipulating Boolean expressions including arithmetic operations, 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 include not only logic operations but also arithmetic operations, such as addition $(+)$, subtraction $(-)$, and multiplication $(\times)$. Each variable is assumed to have a value of either 0 or 1. Any integer is allowed to be used for a constant. Equality $(=)$ and inequality $(<, >, \leqq, \geqq)$ are also 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 and x_2. $(3\times x_1+x_2<4)$ is another example.

When logic operations are applied to integer values other than 0 and 1, we assume that they execute bit-wise logic operations for the binary coded integers, like in many programming languages. Under this modeling, 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 such a function the **Boolean-to-integer (B-to-I) function**.

A sub-part of the arithmetic Boolean expressions also represents a B-to-I function. Therefore, any operation in arithmetic Boolean expressions can be defined as an operation between two B-to-I functions. We can get B-to-I functions for arithmetic Boolean expressions by applying operations on B-to-I functions according to the expressions.

We show an example of obtaining the B-to-I function for the expression $(3\times x_1+x_2<4)$ in Fig. 8.

x1 x2	00	01	10	11
3×x1	0	0	3	3
3×x1 + x2	0	1	3	4
3×x1 + x2 < 4	1	1	1	0

Fig. 8 Computation of arithmetic Boolean expressions.

First, we apply multiplication between a constant function of 3 and one input function of x_1, to obtain the B-to-I function for $(3 \times x_1)$. Then applying addition with x_2, the function for $(3 \times x_1 + x_2)$ is obtained. 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 $(<)$ with the constant function of 4. From the result of computation, we see that this arithmetic Boolean expression is equivalent to the expression $(\overline{x_1} \vee \overline{x_2})$.

3.2 Handling B-to-I Functions Using BDDs

The method of Fig. 8 computes the behavior of B-to-I functions by enumerating all the input combinations. This method is impracticable when there are many input variables since the number of combinations grows exponentially. We show an efficient method of handling B-to-I Functions using BDDs.

As shown in Fig. 9, by encoding an integer with some particular bit length of binary code, a B-to-I function can be decomposed into a number of Boolean functions which represent whether respective bits of the binary code are 1 or 0. These Boolean functions can be represented efficiently using BDDs. Namely, a B-to-I function can be represented by a vector of BDDs.

This method supports only finite values of integers because the bit length should be fixed in advance. If we allocate enough long bits, we will suffer no inconvenience from this constraint. For negative numbers, we use 2's complement representation in our implementation. As the most significant bit is used for the sign bit, the corresponding BDD indicates the condition under which the B-to-I function returns a negative value.

Logic operations, such as AND, OR and EXOR, are implemented as bit-wise operations between the two BDD vectors. Applying BDD operations for respective bits, the result of a new B-to-I function is generated. We defined two kinds of inversion operations. One is bit-wise inversion, and the other is logical inversion, which 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 adopt a simple algorithm of a ripple carry adder, which computes from the lower bit to the higher bit propagating carries. In the same way, other operations, such as subtraction, multiplication, division and shifting can also be composed. Exception handling should be considered for overflow and division by zero.

Positive/negative checking is immediately indicated by the BDD for the sign bit. Using subtraction followed by sign checking, we can compose the comparison operation between two B-to-I functions. This operation generates a new B-to-I function which returns a value of either 1 or 0 to express satisfiability of the equality or inequality.

We can compute the upper or lower bounds of a B-to-I function for all the input combinations. This operation can be composed efficiently based on binary search. To seek the upper bound, we first check whether the function may ever exceed 2^n. If there is a case in which it exceeds 2^n, then we next compare it with $2^n + 2^{n-1}$, or 2^{n-1}. In this way, fixing each bit from the higher to the lower, the upper bound can be computed. The comparison on each bit is composed by BDD operations. The lower bound is found in a similar way.

Computing the upper (lower) bound is defined as a unary operation on B-to-I functions which returns a constant function. This operation can be used conveniently in arithmetic Boolean expressions. For example, the expression:

$$UpperBound\,(F) == F$$

(F is an arithmetic Boolean expression)

gives a function which returns 1 when F has its upper bound value, otherwise returns 0, namely it is the condition to have F maximum.

3.3 Display Formats of B-to-I Functions

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

Integer Karnaugh Maps: Ordinary Karnaugh maps are used to display a matrix of logic values (0/1). Integer Karnaugh maps use integer values for each element, as shown in Fig. 10. This method is helpful to observe the behavior of the B-to-I function. Like

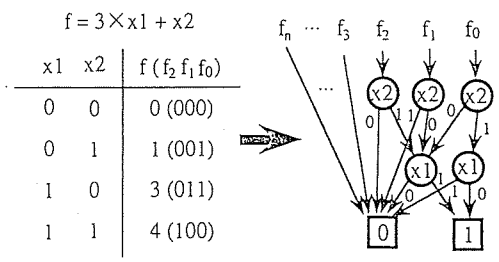


Fig. 9 BDD representation for B-to-I functions.

f = 2×a + 3×b - 4×c + d

f: 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

Fig. 10 An integer Karnaugh map.

±: $(\overline{a}\wedge c\wedge\overline{d})\vee(\overline{b}\wedge c)$
f2: $(a\wedge b\wedge\overline{c})\vee(\overline{a}\wedge c\wedge\overline{d})\vee(b\wedge\overline{c}\wedge d)\vee(\overline{b}\wedge c)$
f1: $(a\wedge\overline{b})\vee(a\wedge d)\vee(\overline{a}\wedge b\wedge\overline{d})$
f0: $(b\wedge\overline{d})\vee(\overline{b}\wedge d)$

Fig. 11 A bit-wise expression.

ordinary Karnaugh maps, they are practicable only for fewer than six input functions. When there are too many inputs, there is a good way to make a matrix for only six input variables and display the upper (lower) bound for the rest of variables.

Bit-wise Expressions: When the B-to-I function is too complex for integer Karnaugh maps, we display the function with a number of Boolean expressions in the sum-of-products format, which represents respective bits of binary coding. This bit-wise format is not so helpful for showing the behavior of the functions as integer numbers, but it allows us to observe the frequency of appearance of an input variable and can estimate a kind of complexity of the functions.

For concise display, we suppress showing the expression of the sign bit if the function never returns negative values. If the function always gives a small value and its higher bits are always zero, it is displayed with zero suppression (omitting showing expressions of '0'). In this reduction rule, a B-to-I function which returns only 1 or 0 is simply displayed by a single Boolean expression. Moreover, a function which returns a constant integer is expressed by a decimal or hex number, not by a bit-wise expression. These reduction rules are applied automatically by checking the BDD representation to be displayed.

Resynthesis of Arithmetic Boolean Expressions: It would be good if we could display the B-to-I function by a simply arithmetic Boolean expression, such as $x_1 + x_2$. Unfortunately, such a method have not been developed yet because it is difficult to extract arithmetic operations from BDD representations and it is not clear what expression is simple. We expect this technique to be related to the extraction of arithmetic functions from logic circuits [14].

3.4 Application to Combinatorial Problems (2)

Using the above method, we can generate BDDs for constraint functions of combinatorial problems given by arithmetic Boolean expressions, and can solve the problems in the way presented in Sect. 2. This method enables us to solve 0-1 linear programming by handling equalities and inequalities directly, without coding complicated procedures in a programming language.

Another feature of using arithmetic Boolean expressions is that we can compute combinatorial problems whose cost function is expressed by nonlinear expressions, whereas the method presented in Sect. 2 is limited to handle linear cost functions. Whether the cost function is linear or not, the upper (lower) bound can be computed by generating BDD representations for the B-to-I functions which represent the cost function explicitly. This approach extends categories of problems to be solved using BDD techniques. However, when the cost function is linear, the method in Section 2 is better since it solves problems without generating BDDs for the cost function.

4. Arithmetic Boolean Expression Manipulator BEM-II

We implemented BEM-II, which generates BDDs of B-to-I functions for arithmetic Boolean expressions and displays them in various formats. This section gives the specifications of BEM-II and its usage.

4.1 Specification

BEM-II has a C-Shell-like interface, both for interactive execution from the keyboard input and for batch jobs from a script file. The program is written in yacc, C, and C++ languages. It runs on 32 bit UNIX machines.

In BEM-II scripts, we describe arithmetic expressions with the kind of variables, *input variables* and *register variables*. Input variables, denoted by strings starting with a lower-case letter, represent the inputs of 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 latter, are used for identifying the memory for saving a temporarily computed B-to-I function. We can describe multilevel expressions using these two types of variables. The results of computation are displayed by irredundant Boolean expressions with input variables only, not including 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 operators such as logical AND, OR, EXOR, NOT, plus, minus, multiply, shift, equality, inequality, and upper/lower bound. The syntax of expressions almost conforms to C language. Neither *If-then-else* nor *while-do* are supported because the system may fail to fetch the next command when the branching condition is given by an expression containing input variables. The list of operators are shown in appendix A.

BEM-II generates BDDs representing B-to-I functions for given arithmetic Boolean expressions. It is enough fast to compute expressions that used to be manipulated by hand. As BEM-II can generate huge BDDs with millions of nodes, limited only by the size of the memory, we can solve large-scale and complicated problems to a degree. The results are displayed in various formats such as integer Karnaugh maps and

bit-wise expressions, as shown in the foregoing sections. An example of interactive execution is given in Appendix B. 1.

4. 2 Performance Evaluation for 8-Queens Problem

We conducted an experiment to solve 8-Queens problem using BEM-II by describing the problem with arithmetic Boolean expressions.

First, we allocated 64 input variables to represent the point on the chessboard matrix. These represent whether or not there is a queen on that points. The constraints that the input variables should satisfy are expressed as follows:

- The sum of 8 variables in the same column is 1.
- The sum of 8 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:

$$Cond_1 = (x_{11} + x_{12} + x_{13} + \cdots + x_{18} == 1)$$

$$Cond_2 = (x_{21} + x_{22} + x_{23} + \cdots + x_{28} == 1)$$

$$\cdots$$

$$Solutions = Cond_1 \wedge Cond_2 \wedge \cdots$$

We show a complete script in Appendix B. 2.

BEM-II feeds the above expressions directly and tries to generate BDDs which represent the set of solutions. If it succeeds in generating BDDs in the main memory, we can immediately find a solution to the problem and count the number of the solutions. (else it may abort). In this way, we can solve the 8-Queens problem by handling abstract forms of the problem.

Table 1 shows the results when we applied this method to the N-Queens problems. The column *#var* shows the number of input variables, and *# BDD* is the number of nodes in the BDDs for representing the set of solutions. We used a SPARC Station 2 (SunOS 4.1.2, 128 MByte).

In the experiments, we solved the problem up to $N = 11$. This shows that BEM-II is less powerful than conventional methods, which have solved up to $N = 15$ using an algorithm based on backtraking and heuristics. This drawback arises from the feature that BEM-II solves all problems in the same way by generating Boolean functions without using specific properties of the problem. There is also another disadvantage we cannot find any solution (not a quasi-best one) when the BDDs cannot be generated because of memory overflow.

However, this method has the great advantage of customizability. Using BEM-II, we can compose scripts for various applications much more easily than developing and tuning a specific program. The script for the 8-Queens problem took only 10 minutes to make. Considering the customizability, we conclude that BEM-II has good computation performance in terms of the total time for programming and execution.

Table 1 Results on N-Queens problems.

N	#var	#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

4. 3 Application for LSI CAD/DA

In researching and developing an algorithm for LSI design systems, we often simulate the algorithm for a small instance to confirm its correctness and efficiency. BEM-II is suitable to such a purpose. It allows us to conduct experiments on algorithms much more efficiently than using hand simulation.

In Appendix B.3, we show a script to solve a *subset sum problem*, that is to find a maximum subset under an upper bound of total cost. It can be solved by 0-1 linear programming. The *0-1 knapsack problem* is described in a similar way. Such problems are often seen in LSI design systems, such as resource scheduling/allocation, logic optimization, and layout.

Using BEM-II, we can solve the subset sum problem by describing a very simple and readable script for BEM-II. We can glance the costs for all the cases by an integer Karnaugh map if the problem is not so large. This is greatly helpuful for analyzing the behavior of expression.

BEM-II is second to well-optimized heuristic algorithms for solving large-scale problems, but it may be utilized as a helpful tool in research and development of LSI design systems.

5. Conclusion

We have presented a method of computing Boolean expressions including arithmetic operations. This method consists of an efficient data structure, manipulation algorithms, and good display formats. BEM-II, implemented based on the above techniques, is customizable for various applications. We expect it to be utilized as a helpuful tool in research and development on digital systems.

BEM-II may abort during computations of large-scale or complicated problems, because it solves all problems in the same way by generating Boolean functions without using the specific properties of the problem. In such cases, it will be good to devise a combination of BDD techniques and well-optimized algorithms developed for a spacific application.

Acknowledgments

The author would like to express his appreciation to Tohru Adachi, Makoto Endo and Atsushi Takahara of NTT LSI laboratories, Toshiaki Miyazaki of NTT Transmission Systems Laboratories and Masayuki Yanagiya of NTT Communication Switching Laboratories for their encouragement.

References

[1] Akers, S. B., "Binary Decision Diagrams," *IEEE Trans. Comput.*, pp. 509-516, 1978.
[2] Bryant, R. E., "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Trans. Comput.*, pp. 677-691, 1986.
[3] Matsunaga, Y., and Fujita, M., "Multi-level Logic Optimization Using Binary Decision Diagrams," *in Proc. ICCAD '89*, pp. 556-559, 1989.
[4] Minato, S., Ishiura, N., and Yajima, S., "Fast Tautology Checking Using Shared Binary Decision Diagram-Benchmark Results," *in Proc. IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, pp. 580-584, 1989.
[5] Lin, Bill, and Somenzi, Fabio, "Minimization of Symbolic Relations," *in Proc. IEEE ICCAD '90*, pp. 88-91, 1990.
[6] Jeong, S.-W., and Somenzi, F., "A New Algorithm for the Binate Covering Problem and its Application to the Minimization of Boolean Relations," *in Proc. IEEE ICCAD '92*, pp. 417-420, 1992.
[7] Minato, S., Ishiura, N., and Yajima, S., "Symbolic Simulation Using Shared Binary Decision Diagram," *1989 IEICE Natl. Conv.*, Rec. IEICE, SA-7-5.
[8] Liaw, H.-T., and Lin, C.-S., "On the OBDD-Representation of General Boolean Functions," *IEEE Trans. Comput.*, pp. 661-664, 1992.
[9] Yajima, S., Ishiura, N., "A Class of Logic Functions Expressible by a Polynomial-Size Binary Decision Diagrams," *in Proc. of the Synthesis and Simulation Meeting and Int. Interchange* (SASIMI '90), 1990.
[10] Minato, S., Ishiura, N., and Yajima, S., "Shared Binary Decision Diagram with Attributed Edgen for Efficient Boolean Function Manipulation," *ACM/IEEE Proc. 27th DAC*, pp. 52-57, 1990.
[11] Fujita, M., Matsunaga, Y., and Kakuda, T., "On Variable Ordering of Binary Decision Diagrams for the Application of Multi-level Logic Synthesis," *in Proc. the European Conference on Design Automation*, pp. 50-54, 1991.
[12] Minato, S., "Minimum-Width Method of Variable Ordering for Binary Decision Diagrams," *IEICE Trans. Fundamentals*, vol. E75-A, no. 3, pp. 392-399, 1992.
[13] Minato, S., "Fast Generation of Irredundant Sum-of-Products Forms from Binary Decision Diagrams," *in Proc. of the Synthesis and Simulation Meeting and Int. Interchange* (SASIMI '92, Japan), pp. 64-73, 1992.
[14] Ohmura, M., Yasuura, H., and Tamaru, K., "Extraction of Functional Information from Combinational Ciacuits," *in Proc. IEEE ICCAD '90*, pp. 176-179, 1990.

Appendix A: List of Operators

The syntax of arithmetic Boolean expressions conforms to C language. The followings are the available operators in the order of priority.

```
( )
[ ]
!  ~  +  - (unary)
*
+  - (binary)
<<  >>
<  <=  >  >=
==  !=
&
^
|
? :
UpperBound( )  LowerBound( )
```

F [G] returns F reduced with a *don't care* condition of G=0. ! is logical inversion. ! F returns 1 when F=0, else returns 0. ~ is bit-wise inversion. It is different to !. ~0 returns −1. F?G:H returns G when F=1, else returns H.

Appendix B: Examples of Execution

B. 1 Interactive Mode

```
% bemII
***** BEM-II: Boolean Expression Manipulator II (Ver. 3.2) *****
bemII> symbol a(3) b(2) c(4) d(1)
bemII> Sum = a*3 + b*4 + c*5 - d*2
bemII> print Sum

 +-: !a & !b & !c & d
   : :..
  3: a & b & c | a & c & !d | !a & !b & !c & d | b & c & !d
  2: b ^ ( a & c & d | !a & c & !d | !a & !c & d )
  1: d ^ ( a & !c )
  0: a ^ c
bemII> print /map Sum
  a b : c d
      |    00    01    11    10
   00 |     0    -2     3     5
   01 |     4     2     7     9
   11 |     7     5    10    12
   10 |     3     1     6     8
bemII> print UpperBound(Sum)
    12
bemII> print LowerBound(Sum)
    -2
bemII> print Sum > 6
   a & b & !d | a & c & !d | b & c
bemII> F = Sum > 6
bemII> print /mincover F
<Positive>: a b
bemII> print /mincost F
   5
bemII> exit
%
```

B. 2 8-Queens Problem

Script for Input

```
####### 8-Queens Problem #######
symbol a00 a10 a20 a30 a40 a50 a60 a70
symbol a01 a11 a21 a31 a41 a51 a61 a71
symbol a02 a12 a22 a32 a42 a52 a62 a72
symbol a03 a13 a23 a33 a43 a53 a63 a73
symbol a04 a14 a24 a34 a44 a54 a64 a74
symbol a05 a15 a25 a35 a45 a55 a65 a75
symbol a06 a16 a26 a36 a46 a56 a66 a76
symbol a07 a17 a27 a37 a47 a57 a67 a77

X0 = (a00 + a10 + a20 + a30 + a40 + a50 + a60 + a70 == 1)
X1 = (a01 + a11 + a21 + a31 + a41 + a51 + a61 + a71 == 1)
X2 = (a02 + a12 + a22 + a32 + a42 + a52 + a62 + a72 == 1)
X3 = (a03 + a13 + a23 + a33 + a43 + a53 + a63 + a73 == 1)
X4 = (a04 + a14 + a24 + a34 + a44 + a54 + a64 + a74 == 1)

X5 = (a05 + a15 + a25 + a35 + a45 + a55 + a65 + a75 == 1)
X6 = (a06 + a16 + a26 + a36 + a46 + a56 + a66 + a76 == 1)
X7 = (a07 + a17 + a27 + a37 + a47 + a57 + a67 + a77 == 1)

Y0 = (a00 + a01 + a02 + a03 + a04 + a05 + a06 + a07 == 1)
Y1 = (a10 + a11 + a12 + a13 + a14 + a15 + a16 + a17 == 1)
Y2 = (a20 + a21 + a22 + a23 + a24 + a25 + a26 + a27 == 1)
Y3 = (a30 + a31 + a32 + a33 + a34 + a35 + a36 + a37 == 1)
Y4 = (a40 + a41 + a42 + a43 + a44 + a45 + a46 + a47 == 1)
Y5 = (a50 + a51 + a52 + a53 + a54 + a55 + a56 + a57 == 1)
Y6 = (a60 + a61 + a62 + a63 + a64 + a65 + a66 + a67 == 1)
Y7 = (a70 + a71 + a72 + a73 + a74 + a75 + a76 + a77 == 1)

Z1 = (a10 + a01 < 2)
Z2 = (a20 + a11 + a02 < 2)
Z3 = (a30 + a21 + a12 + a03 < 2)
Z4 = (a40 + a31 + a22 + a13 + a04 < 2)
Z5 = (a50 + a41 + a32 + a23 + a14 + a05 < 2)
Z6 = (a60 + a51 + a42 + a33 + a24 + a15 + a06 < 2)
Z7 = (a70 + a61 + a52 + a43 + a34 + a25 + a16 + a07 < 2)
Z8 = (a71 + a62 + a53 + a44 + a35 + a26 + a17 < 2)
Z9 = (a72 + a63 + a54 + a45 + a36 + a27 < 2)
Za = (a73 + a64 + a55 + a46 + a37 < 2)
Zb = (a74 + a65 + a56 + a47 < 2)
Zc = (a75 + a66 + a57 < 2)
Zd = (a76 + a67 < 2)

W1 = (a06 + a17 < 2)
W2 = (a05 + a16 + a27 < 2)
W3 = (a04 + a15 + a26 + a37 < 2)
W4 = (a03 + a14 + a25 + a36 + a47 < 2)
W5 = (a02 + a13 + a24 + a35 + a46 + a57 < 2)
W6 = (a01 + a12 + a23 + a34 + a45 + a56 + a67 < 2)
W7 = (a00 + a11 + a22 + a33 + a44 + a55 + a66 + a77 < 2)
W8 = (a10 + a21 + a32 + a43 + a54 + a65 + a76 < 2)
W9 = (a20 + a31 + a42 + a53 + a64 + a75 < 2)
Wa = (a30 + a41 + a52 + a63 + a74 < 2)
Wb = (a40 + a51 + a62 + a73 < 2)
Wc = (a50 + a61 + a72 < 2)
Wd = (a60 + a71 < 2)

C = 1
C = C & X0 & X1 & X2 & X3 & X4 & X5 & X6 & X7
C = C & Y0 & Y1 & Y2 & Y3 & Y4 & Y5 & Y6 & Y7
C = C & Z1 & Z2 & Z3 & Z4 & Z5 & Z6 & Z7 & Z8 & Z9 & Za & Zb & Zc & Zd
C = C & W1 & W2 & W3 & W4 & W5 & W6 & W7 & W8 & W9 & Wa & Wb & Wc & Wd


print /size C
print /count C
print /mincover C
```

Result of Execution

```
% bemII queen8.bem
   2450 (3014)
   92
<Positive>: a77 a36 a05 a24 a53 a12 a61 a40
%
```

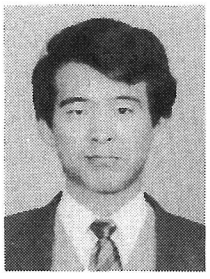
B. 3 Subset Sum Problem

Script for Input

```
##### SUBSET-SUM Problem #####
symbol a b c d e f

Sum = 2*a + 3*b + 4*c + 3*d + 5*e + 6*f
print /map Sum

S = Sum * (Sum <10)
print /map S

C = UpperBound(S)
print C
print (C == S)
```

Result of Execution

```
% bemII subsetsum.bem
 a b c : d e f
       |   000   001   011   010 |   110   111   101   100
 000   |     0     6    11     5 |     8    14     9     3
 001   |     4    10    15     9 |    12    18    13     7
 011   |     7    13    18    12 |    15    21    16    10
 010   |     3     9    14     8 |    11    17    12     6
       |                         |
 110   |     5    11    16    10 |    13    19    14     8
 111   |     9    15    20    14 |    17    23    18    12
 101   |     6    12    17    11 |    14    20    15     9
 100   |     2     8    13     7 |    10    16    11     5
 a b c : d e f

       |   000   001   011   010 |   110   111   101   100
 000   |     0     6     0     5 |     8     0     9     3
 001   |     4     0     0     9 |     0     0     0     7
 011   |     7     0     0     0 |     0     0     0     0
 010   |     3     9     0     8 |     0     0     0     6
       |                         |
 110   |     5     0     0     0 |     0     0     0     8
 111   |     9     0     0     0 |     0     0     0     0
 101   |     6     0     0     0 |     0     0     0     9
 100   |     2     8     0     7 |     0     0     0     5
     9
 a & b & c & !d & !e & !f | a & !b & c & d & !e & !f | !a & b & !c & !d &
    !e & f | !a & !b & c & !d & e & !f | !a & !b & !c & d & !e & f
%
```



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