Generation of BDDs from Hardware Algorithm Descriptions

Shin-ichi Minato
NTT System Electronics Laboratories
Kanagawa Pref., 243-01, Japan

Abstract

We propose a new method for generating BDDs from hardware algorithm descriptions written in a programming language. Our system can deal with control structures, such as conditional branches (if-then-else) and data dependent loops (while-end). Once BDDs are generated, we can immediately check the equivalence of two different algorithm descriptions just by comparing BDDs. This method can also be applied to verification between algorithm-level and gate-level designs. Another interesting application is to synthesize loop-free logic circuits from algorithm descriptions. We show the experimental results for some practical examples, such as Greatest Common Divisor (GCD) calculation. Although our method has a limitation in size of problems, it is very practical and useful for actual design verification.

1 Introduction

In designing hardware algorithms, one may write the register transfer (RT) level or gate-level HDL directly, but many designers prefer to use a programming language, such as C, Pascal, etc., since they are familiar with those languages. We call this *algorithm-level description*. Hardware designers may produce several versions of algorithm-level descriptions, one of which may represent just a function of the submodule independent of the real implementation, while another may give the best one in terms of computation time or hardware resource's requirements.

These hardware algorithms are usually composed by hand, so design errors may possibly occer. Although many verification tools are available for the RT-level or gate-level descriptions[1, 2], problems still remain in verifying the correctness of algorithm-level ones. Currently, most designers write an algorithm as carefully as possible, and confirm the correctness by software simulation with a number of stimuli. Unfortunately, it is hard to cover all instances, so unexpected bugs may possibly remain. Recently, Srivas et al.[3] presented a formal verification system based on the automatic theorem prover; however, in this system, users are sometimes required to give additional information to prove the theorem efficiently, and it is hard for many designers to master.

In this paper, we propose a new method for generating BDDs from algorithm-level descriptions written in a programming language. Our method can deal with control structures, such as conditional branches (if-then-else), data dependent loops (while-end), and arrays of variables. Once BDDs are generated, we can immediately check the equivalence of two different algorithms just by comparing BDDs, since BDDs are canonical representations of logic functions. This method can also be applied to verification between algorithm-level and RT-level designs. Another interesting application is to synthesize loop-free logic circuits from BDDs which have been extracted from the algorithm-level descriptions.

Based on this method, we implemented a symbolic execution system for algorithm-level descriptions. This system can be utilized for many practical problems; for example, we tried *Euclid's algorithm*[4] that calculates the Greatest Common Divisors (GCDs), and succeeded in generating BDDs which gave GCDs for any pair of integers up to a thousand. We also generated BDDs from another algorithm of GCD calculation, and checked the equivalence of the two BDDs for the different algorithms. In addition, our system is very convenient for analyzing the behavior of the algorithm; for example, we can check the maximum times of loop execution, and which input assignments lead to it. A stable state in the endless loop can also be detected. Although our method has a limitation in size of problems, it is very practical and useful for actual design verification.

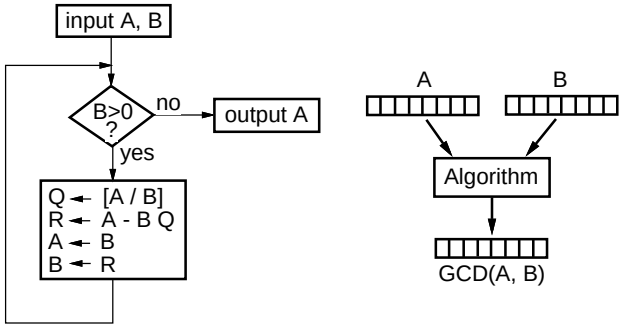
In the remainder of this paper, we first show an example of algorithm-level description and the technical problems of generating BDDs from it. We then present the data structure and algorithm for calculating arithmetic and Boolean expressions. Next, we propose a method for handling control structures such as branches and loops. Manipulating arrays of variables is also discussed. Finally, we describe the implementation of our system and show experimental results.

2 Algorithm-Level Descriptions

First, we clarify the problem discussed in this paper. As an example, Fig. 1(a) shows a flow chart of Euclid's Algorithm to calculate GCDs for a given pair of integers. It consists of the arithmetic operations of integers, assignments of variables, a conditional branch, and a loop structure. For given inputs A and B, this algorithm repeats division operations until $B = 0$, and then output the GCD.

In case of implementing this algorithm through hardware, we usually design a sequential machine with some registers, ALUs, and a control unit. However, if we only consider the relation between input and output, this algorithm can be regarded as a logic func-

ICCAD '96
1063-6757/96 $5.00



(a) Flow chart. (b) Function of algorithm.

Figure 1: Euclid's algorithm for GCD calculation.

tion that computes a binary-coded vector GCD(A, B) from A and B as shown in Fig. 1(b). The goal of this paper is to generate BDDs representing such logic functions based on symbolic execution. As BDDs are canonical representations of logic functions, the forms of BDDs are independent of the internal structures of algorithms. By comparing these BDDs, we can check the equivalence of two algorithms. If the BDD forms are different, the logical EX-OR of the two BDDs gives the counter examples of inputs to make different outputs.

In this paper, we consider the algorithm-level descriptions having the following structures, which are commonly supported in many programming languages.

- Expressions containing arithmetic operations of integers, equality and inequality, and bit-wise logic operations.
- Program variables for saving the results of calculation to be referred to in other expressions.
- Conditional branches (if-then-else)
- Data dependent loops (while-end).
- Arrays of variables.

Discussed theoretically, *algorithms* may have an infinite number of instances. In such cases, we cannot generate BDDs since they require an infinite number of input variables. In this paper, we assume a fixed number of input/output variables with a finite bit-length.

In generating BDDs based on the symbolic execution of algorithms, there are two technical problems:

- Data structures for symbolic calculation of arithmetic operations of integers.
- Symbolic execution of data dependent branches and loops. (For example, repeating times depends on the values of symbolic inputs.)

We have already presented a method using *BDD vectors*[5] for calculating arithmetic operations, but it does not handle data dependent branches and loops. In the following sections, we present methods of solving these problems.

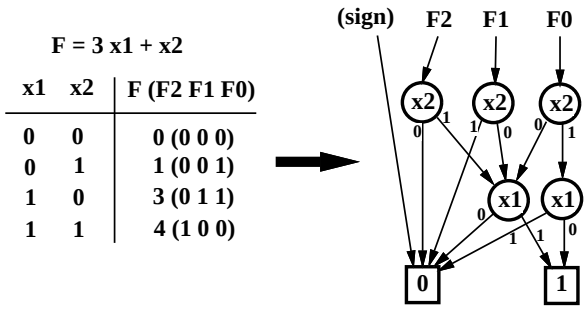


Figure 2: BDD vector for a B-to-I function.

3 Manipulation of BDD Vectors

We present a method for manipulating BDD vectors for calculating arithmetic and logic operations of integers. (Details are described in [5].) In this method, we deal with *arithmetic Boolean expressions* and *B-to-I (Boolean-to-integer) functions*, which are extended models of conventional Boolean expressions and switching functions.

Arithmetic Boolean expressions are extended Boolean expressions including not only logic operators but also arithmetic operators for constant integers and switching input variables (symbols). For example, $(3x_1+x_2)$ is an arithmetic Boolean expression, that returns an integer value 0, 1, 3, or 4 for each assignment of $\{0, 1\}$ to the input variables x_1, x_2. In general, an arithmetic Boolean expression represents a function from a binary-vector to an integer: $(B^n \rightarrow I)$. We call this a *B-to-I (Boolean-to-integer) function.*

In order to represent B-to-I functions using BDDs, we decompose a B-to-I function into n pieces of switching functions each of which represents a digit of the binary-corded integer, as shown in Fig. 2. We call such representation the *BDD vector.* For negative numbers, we used 2's complement representation. The BDD of the most significant bit indicates the conditions under which the function returns negative value.

In our method, an arithmetic expressions can be assigned to a program variable to save the calculation result and reuse it in subsequent expressions. Each program variable stands for a B-to-I function, and it is represented by a BDD vector. In this paper, we denote the program variables by strings starting with an uppercase letter, to distinguish them from input variables denoted by strings starting with a lowercase letter.

When the algorithm-level description requires an input variable having integer values, we represent it by using a number of switching input variables for each digit of binary code, as

$$X = x_1 + 2x_2 + 4x_2 + \cdots + 2^{n-1}x_n.$$

For generating a BDD vector for a given arithmetic Boolean expression, we first generate trivial ones representing input variables or constant numbers, and then construct BDD vectors by applying some arithmetic and logic operations according to the structure of the expression. Arithmetic operations like addition,

```
A ← 0 ;  B ← 0
if ( 3x + y < 4 )  then
    A ← 10
    B ← 3
else
    A ← 5
endif
```

(a) Original description.

	x y = 00	01	10	11
A	10	10	10	5
B	3	3	3	0

(b) Expected result.

```
A ← 0 ;  B ← 0
C ← (3x + y < 4)
E ← C
A ← (E · 10) + (¬E · A)
B ← (E · 3) + (¬E · B)
E ← ¬C
A ← (E · 5) + (¬E · A)
```

(c) Translated description.

Figure 3: An example of if-then-else structure.

subtraction, multiplication, division, and shifting can be performed by using BDD logic operations simulating a conventional hardware algorithm of arithmetic circuits. We assumed integer values only, so division produces a quotient and a remainder. Equality and inequality operation is performed by subtraction followed by evaluation of the sign-bit BDDs. Logic operations, such as AND, OR, and EXOR, are implemented as bit-wise operations between two BDD vectors.

It is very useful for practical applications to find the upper (or lower) bound value of a B-to-I function for all possible combinations of input values. This can be done efficiently by determining each digit of BDD from the highest to the lowest based on the binary search.

4 Handling of Control Structures

It is a problem to generate BDDs from the descriptions including data dependent branches and loops because we hardly determine which statement will be executed, or how many times the loop will be repeated. For example, the Eucrid's algorithm (Fig. 1) repeats the loop until $B = 0$, but the repeat times depends on the values of input variables.

Here, we present a way to handle the two control structures: *if-then-else* and *while-end* having symbolic conditions. These two structures are basic enough to describe the generality of algorithms.

4.1 If-Then-Else Structures

First, we describe a method for dealing with the if-then-else structures with symbolic conditions. Figure 3(a) shows a simple example of such a structure. In this case, the condition $(3x + y < 4)$ is satisfied when $x = 0$ or $y = 0$. If it is satisfied, then the assignment $A \leftarrow 10$ and $B \leftarrow 3$ are executed. Otherwise, A becomes 5 but B keeps its initial value of 0. After executing this if-then-else part, A and B should have the values depending on x and y, as shown in Fig. 3(b).

For computing the B-to-I functions from such a data dependent branch, we defined the switching function $\mathcal{E}$, that represents whether the statement is executable or not. We call this *executable function*. Using an executable function, the assignment

$$A \leftarrow expr$$

can be translated into

$$A \leftarrow (\mathcal{E} \cdot expr) + (\overline{\mathcal{E}} \cdot A),$$

which means that the value of A does not change when $\mathcal{E} = 0$. In this way, the description of Fig. 3(a) can be translated into a deterministic sequence as shown in Fig. 3(c). We can generate BDD vectors from the translated descriptions.

When the if-then-else structures are nested, the second depth of executable function becomes the conjunction with the first depth ones. In general, the n-th depth of executable function $\mathcal{E}_n$ can be computed as the conjunction of $\mathcal{E}_{n-1}$ and the function of the n-th condition. We define $\mathcal{E}_0 = 1$. In our implementation, we prepared a stack to save $\mathcal{E}_n$ for each depth of the if-then-else structures.

Using the executable functions, we can eliminate data dependent branches from the algorithm descriptions. Multiple branches, such as the *case* statement, can be translated into a number of nesting if-then-else structures.

```
A ← 3x + y ;  B ← 0
while (A > 0)
    B ← B + A
    A ← A − 1
end
```

(a) Original description.

	x y = 00	01	10	11
A	0	0	0	0
B	0	1	6	10

(b) Expected result.

```
A ← 3x + y ;  B ← 0
E ← (A > 0)
while ( E ≢ 0 )
   B ← E·(B + A) + (¬E · B)
   A ← E·(A − 1) + (¬E · A)
   E ← E · (A > 0)
end
```

(c) Translated description.

Figure 4: An example of while-end structure.

4.2 While-End Structures

Next, we propose a way to deal with the while-end structures with symbolic conditions. We show an example in Fig. 4(a). In this case, the assignments $B \leftarrow B + A$ and $A \leftarrow A - 1$ are repeated while $(A > 0)$ is satisfied. The repetitions depend on the values of x and y. The final results of A and B are shown in Fig. 4(b).

The concept of the executable function is also useful in executing the while-end structures. If the *while-*

```
I ← 10
while ( I > 0 )
    I ← I − 2x
end
```
(a) Detectable case.

```
I ← 10
while ( I > 0 )
    I ← I + 1
end
```
(b) Undetectable case.

Figure 5: Detection of endless loops.

condition is not satisfied at the first evaluation, the loop is never executed. Therefore, the first time, the statements in the loop are executed under executable function $\mathcal{E}$ that corresponds to the while-condition. After the first execution, the values of some program variables may change, and the while-condition is evaluated again. In this step, the new executable function can be computed as the conjunction of current $\mathcal{E}$ and the new while-condition. We repeat this procedure until $\mathcal{E} \equiv 0$ (unsatisfiable). This unsatisfiability checking is quite easily done by BDD operations. Unless the algorithm description has endless loops, $\mathcal{E}$ eventually becomes unsatisfiable, and we can break the loop. Using this method, the example in Fig. 4(a) can be translated into a deterministic procedure, as shown in Fig. 4(c).

The nested loops can be executed in a similar way to the if-then-else structures. The n-th depth of executable function $\mathcal{E}_n$ is initialized as $\mathcal{E}_{n-1}$, and we then compute the conjunction of the while-condition. As usual programming languages, the number of total repeating times becomes the product of the repeating times for each depth. The nesting of if-then-else and while-end structures is also executed similarly.

If the description has an endless loop, our method will not terminate. A way to detect the existence of an endless loop. Every time we evaluate the while-condition, we check the programming variables which appears in the loop. If all the variables have kept the same values as the last time, we can detect a stable state to allow the loop to be continued endlessly. The final executable function represents the condition of the input variables to fall into the endless loop. This information is very useful in debugging the algorithm description. We show an example in Fig. 5(a). In this case, the loop does not terminate when $x = 0$, and it can be detected with our method.

Unfortunately, there is another type of endless loops which cannot be detected. Figure 5(b) is one such example. Variable I increases every time, but it will never break the while-condition. It is very difficult to recognize this type of endless loops.

4.3 Arrays of Variables

When describing algorithms with loop structures, we sometimes use arrays of variables. It makes the descriptions more compact, and increases the applicability of the programming language. However, there is a problem in how to manipulate the array specified by a symbolic index. For example, $A(x+y)$ refers $A(0), A(1)$, and $A(2)$ depending on the values of x and y. Namely, the statement:

$B \leftarrow A(x+y)$

is equivalent to:

$B \leftarrow \overline{x} \cdot \overline{y} \cdot A(0) + (x \oplus y) \cdot A(1) + x \cdot y \cdot A(2)$,

and

$A(x+y) \leftarrow B$

is equivalent to the sequence of statements:

if $(\overline{x} \cdot \overline{y})$ then $A(0) \leftarrow B$,
if $(x \oplus y)$ then $A(1) \leftarrow B$,
if $(x \cdot y)$ then $A(2) \leftarrow B$.

In our implementation, such expressions are translated automatically. The space for the array variables are allocated when they appear for the first time. The indices of negative numbers can also be used. For example, when the expression $A(100x - 50y)$ appears, the space for only four variables $A(0), A(50), A(100)$, and $A(-50)$ are allocated. (no space allocated for $A(1) \cdots A(49)$.) The possible values of the index can be known by checking BDD vectors.

5 Implementation and Experimental Results

Based on the above method, we implemented a symbolic execution system for algorithm-level descriptions. The program is written in C++ language on a SPARC station 10 (SunOS 4.1.3, 128MB). This system is implemented as an interpreter with a C-shell-like interface, both for interactive execution by keyboard and for batch jobs to read a script file. It computes the B-to-I functions for the arithmetic Boolean expressions, and the results are represented by BDD vectors.

Our system allows up to 65,535 different input variables. We can generate up to several millions of BDD nodes, limited by main memory size. Using `print` command, any time we can observe the current results of execution. The results of B-to-I functions are displayed in various formats, such as Karnough maps, bit-wise Boolean expressions, graphic display of BDD forms, and net lists in Verilog HDL. The statistical informations, such as number of BDD nodes, can also be displayed. These formats are helpful in understanding the behavior of the algorithms.

For evaluating the performance of our system, we conducted experiments to generate BDD vectors from some practical examples of algorithm-level descriptions. The results are shown in Table 1. The column "Name" indicates the sort of functions, "#In" and "#Out" shows the total number (bit) of the inputs and outputs. "BDD node" shows the number of nodes for representing output variables, not including ones for intermediate variables.

In this table, the name "euc-n" and "mygcd-n" are the scripts of GCD calculation for a pair of n-bit integers. "prime-n" means the prime check function, which returns 1/0 whether the given n-bit integer is a prime number or not. "hamm-n" describes the function that counts the Hamming distance between a pair of n-bit binary codes. "fact-n" calculates the factorials for given n-bit integers. For "fact-6", the BDD vector represents all factorials up to 63 !, and the output bit-length reaches as many as 290. "sel-n" describes the function of the n-to-1 data selector, that has n-bit data inputs and $\lceil \log n \rceil$-bit control inputs. "enc-n" is the n-bit priority encoder from one-hot code into

Table 1: Experimental results.

Name	#In	#Out	BDD node*	Time(s)
euc-4	8	4	86	1.2
euc-6	12	6	775	7.8
euc-8	16	8	6,850	154.6
euc-10	20	10	63,589	3,906.5
mygcd-4	8	4	86	7.4
mygcd-6	12	6	775	57.3
mygcd-8	16	8	6,850	1,089.1
prime-8	8	1	46	3.6
prime-12	12	1	352	36.0
prime-16	16	1	3,242	880.0
hamm-15	30	4	387	0.3
hamm-31	62	5	1,667	1.4
hamm-63	126	6	6,915	8.0
fact-4	4	41	65	2.6
fact-6	6	290	1,160	170.3
sel-16	20	1	31	0.2
sel-64	70	1	127	0.9
sel-256	264	1	511	3.4
enc-15	15	4	42	0.3
enc-63	63	6	290	1.6
enc-255	255	8	1,666	14.2

* using negative edges.

binary code.

Here we illustrate the scripts and BDD forms for several examples. "euc-n" is the description of Euclid's algorithm for n-bit GCD calculation. Figure 6(a) shows the script "euc-4". (In this example, we assumed that the output is zero when at least one of the inputs is zero.) Our system can interpret this script directly, and generates a BDD vector including all the information about the GCDs for any pair of integers. The result of BDD form is printed out in Fig. 7. If we assign certain 0/1 values into the input variables, this BDD vector immediately gives a binary-coded GCD number. We succeeded in generating the BDD vector for "euc-10" (up to 1,023) in a practical time and space.

We also tried to run "mygcd-n", which is another algorithm for GCD calculation. The script "mygcd-4" is shown in Fig. 6(b). This algorithm repeats checking for all the numbers whether each number can be a common divisor or not. Our system interprets this description as well. The result of BDD vector was completely the same as one for Eucrid's algorithm. In terms of CPU time, "euc-n" is much faster than "mygcd-n" because of the difference in total execution steps.

For debugging or analyzing the descriptions, we can insert additional program variables to observe the internal behavior of execution. For example, Euclid's algorithm repeats the loop many times, but the maximum repeat times are not obvious. By inserting the statement `I=I+1` at the inside of while-loop and looking at the result of `I`, we can check the maximum repeat times and which input assignments lead to it.

In Fig. 8, we show another script example, which describes 8-bit prime check function "prime-8". This

```
symbol a3 a2 a1 a0
symbol b3 b2 b1 b0
A = a0+2*a1+4*a2+8*a3
B = b0+2*b1+4*b2+8*b3
if(A==0 | B==0) then
  A = 0
else
  while(B > 0)
    Q = A/B
    R = A - B*Q
    A = B
    B = R
  end
endif
print /size A
```

(a) Eucrid's algorithm.

```
symbol a3 a2 a1 a0
symbol b3 b2 b1 b0
A = a0+2*a1+4*a2+8*a3
B = b0+2*b1+4*b2+8*b3
if(A==0 | B==0) then
 A = 0
else
 C = 1
 I = 2
 while(I<=A & I<=B)
  while(A==A/I*I & B==B/I*I)
   C = C*I
   A = A/I
   B = B/I
  end
  I = I+1
 end
endif
print /size C
```

(b) "mygcd" algorithm.

Figure 6: Scripts for 4-bit GCD calculation.

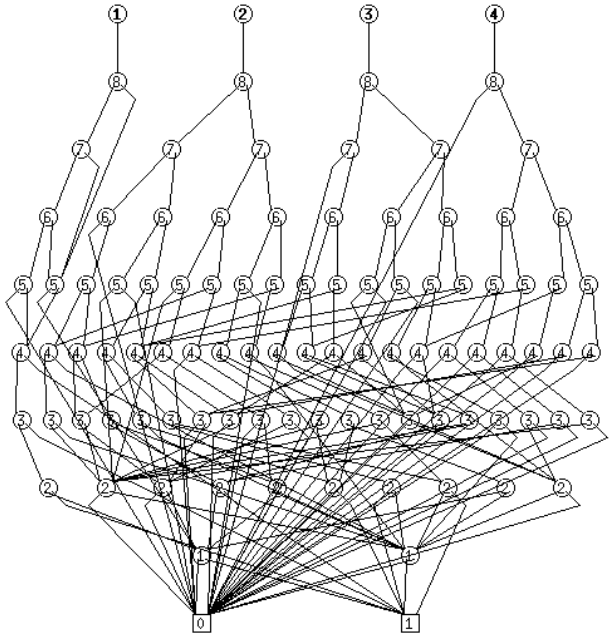


Figure 7: BDDs for 4-bit GCD calculation.

algorithm attempts to divide the given integer by all possible numbers, and if no number can be a divisor, the output `F` will be true. If we implement this algorithm as a sequential machine with some registers and ALUs, it will take many clock cycles to compute results. Our system can flatten this algorithm into a combinational logic. The result of BDD form is shown in Fig. 9. This BDD corresponds to a loop-free combinational circuit that performs prime checking within a single clock cycle.

The number of BDD nodes does not directly depend on the number of execution steps in the scripts. It is decided by the number of input variables and the nature of the functions. For the tractable functions, we can use up to several hundreds of input variables;

```
symbol a(8..1)
A = 0; I = 8
while(I>0)
  A = A*2+a(I)
  I = I-1
end
F = 1
if(A<=1) then
  F = 0
endif
if(A%2==0 & A>2) then
  F = 0
endif
I = 3
while(I*I<=A & F==1)
  if(A%I==0) then
    F = 0
  endif
  I = I+2
end
print /size F
```

Figure 8: Scripts for 8-bit prime check function.

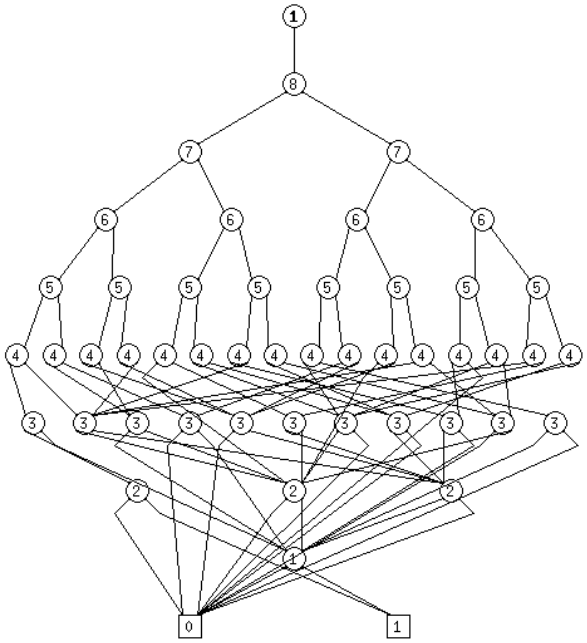


Figure 9: BDDs for 8-bit prime check function.

however, in the worst case, the BDD size grows exponentially to the number of input variables. It is a limitation of our method in size of problems. In terms of execution time, it depends not only on the BDD size but also on the number of the execution steps. Deeply nested loops are not favorable since they are time consuming.

6 Conclusion

We have proposed a new method for generating BDDs from algorithm-level descriptions written in a programming language. Our system can handle conditional branches and data dependent loops. Once BDDs are generated, we can immediately check the equivalence of two different algorithm descriptions just by comparing BDDs. In addition, our system can display the current results of internal program variables. It is very useful for debugging and analyzing the behavior of algorithms.

Another interesting application is to synthesize loop-free logic circuits from algorithm-level descriptions. We previously developed a method[6] to synthesize gate-level combinational circuits from BDD representations. This method would enable us to synthesize hardware modules from algorithm-level descriptions. FPGAs are useful for implementing such circuits into actual devices. In this way, we can accelerate the system by transforming a part of software into a hardware module that computes the outputs within a single clock cycle.

So far, our system does not support subroutine calls. One of the reasons is that we implemented the system as an interpreter with both interactive and batch-style interfaces. If we implement another system which runs in batch-style only, it is possible to handle the subroutines with local scope variables.

Algorithm-level descriptions written in a programming language are very easy to read and write for most of designers. Our system can interpret such descriptions directly. Although there is a limitation in size of problems, our method is very practical and useful for actual design verification.

References

[1] R. Bryant and Y.-A. Chen, Verification of Arithmetic Circuits with Binary Moment Diagrams, *Proc. of ACM/IEEE 32rd DAC*, pp. 535-541, June, 1995.

[2] J. Burch, E. Clarke, L. McMillan, and D. Dill, Sequential Circuit Verification Using Symbolic Model Checking, *Proc. of ACM/IEEE 27th DAC*, pp. 535-541, June, 1990.

[3] M. Srivas and A. Miller, Applying Formal Verification to a Commercial Microprocessor, *Proc. of IEEE CHDL'95*, pp. 493-502, Aug., 1995.

[4] D. Knuth, The Art of Computer Programming, Vol. I: Fundamental Algorithms, *Addison-Weslay, Reading Mass.*, 1973

[5] S. Minato, Arithmetic Boolean Expressions, In Binary Decision Diagrams and Application for VLSI CAD, chapter 9, pp. 109-128, *Kluwer Academic Publishers*, Oct., 1995.

[6] S. Minato, Fast Factorization Method for Implicit Cube Set Representation, *IEEE Trans. on CAD*, Vol. 15, No. 4, pp. 377-384, Aplil, 1996.

[7] S. Minato, Graph-Based Representations of Discrete Functions, In T. Sasao, editor, Representation of Discrete Functions, chapter 1, pp. 1-27, *Kluwer Academic Publishers*, April, 1996.