# Implicit Manipulation of Polynomials Using Zero-Suppressed BDDs

**Shin-ichi Minato**
**NTT LSI Laboratories**
**3-1 Morinosato-Wakamiya, Atsugi-shi, Kanagawa Pref., 243-01 Japan**

## Abstract

*We present a new technique that broadens the scope of BDD application. It involves manipulating arithmetic polynomials containing higher-degree variables and integer coefficients. Our method can represent large-scale polynomials compactly and uniquely, and it greatly accelerates computation of polynomials. As the polynomial calculus is a basic model in mathematics, our method is very useful in various areas, including formal verification techniques for VLSI design.*

## 1 Introduction

Recently, Binary Decision Diagrams (BDDs), graph-based representations of Boolean functions[1], have attracted much attention because they enable us to manipulate Boolean functions efficiently in terms of time and space. There are many cases in which conventional algorithms can be significantly improved by using BDDs[2][3].

As our understanding of BDDs has deepened, the range of applications has broadened. Besides Boolean functions, we are often faced with manipulating *sets of combinations* in many LSI design problems. By mapping a set of combinations into the Boolean space, it can be represented as a characteristic function using a BDD. This method enables us to manipulate a huge number of combinations implicitly, which has never been practical before. Based on implicit set representation, new two-level logic minimization methods have been developed[4][5]. These techniques are also used to solve a kind of covering problem[6].

A *zero-suppressed BDD* (0-sup-BDD)[7] is a new type of BDD adapted the implicit set representation. It can manipulate sets of combinations more efficiently than conventional BDDs, especially when dealing with sparse combinations. We have recently studied cube set algebra for manipulating sets of combinations[8], and proposed efficient algorithms for computing cube set operations based on 0-sup-BDDs. This technique is useful for many practical activities related to LSI design, including multi-level logic synthesis[9] and fault simulation.

In this paper, we present a new technique that broadens the scope of BDD application. It involves manipulating arithmetic polynomial formulas containing higher-degree variables and integer coefficients. Using 0-sup-BDDs, we can represent large-scale polynomials compactly and uniquely. We developed efficient algorithms for polynomial calculus based on 0-sup-BDD operations. In this method, we can flatten arithmetic expressions into canonical forms of polynomials with millions of terms, which have never been represented before. Constructing canonical forms of polynomials immediately leads to equivalence checking of arithmetic expressions. Since polynomial calculus is a basic model in mathematics, our method is expected to be useful for various problems.

In this paper, we first explain 0-sup-BDDs and their operations. We then present a method for representing polynomials with 0-sup-BDDs, and discuss the operation algorithms for polynomial calculus. Finally, we show implementation of our method and the application for LSI CAD.

## 2 Zero-Suppressed BDDs

*Zero-suppressed BDDs (0-sup-BDDs)*[7] are a new type of BDD[7] adapted for representing sets of combinations. They are based on the following reduction rules:

- Eliminate all nodes with the 1-edge pointing to the 0-terminal node. Then connect the edge to the other subgraph directly (Fig. 1).
- Share all equivalent sub-graphs in the same manner as with conventional BDDs.

Notice that, contrary to conventional BDDs, we do not eliminate nodes whose two edges both point to the same node. This reduction rule is asymmetric for the two edges because the nodes remain when their 0-edge points to a terminal node. When the number and order of the variables are fixed, 0-sup-BDDs provide canonical forms for Boolean functions.

Figure 2 illustrates conventional and 0-sup-BDDs representing sets of combinations. Using the "0-sup"

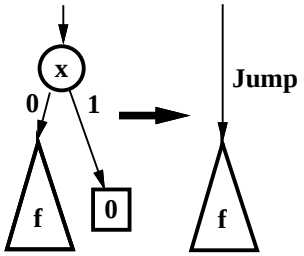
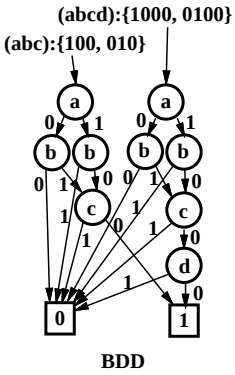



Fig. 1: Reduction Rule for 0-sup-BDDs



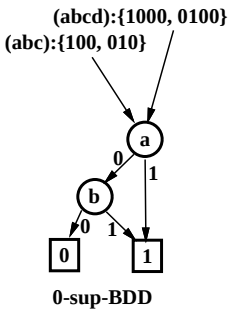




Fig. 2: Effect of "0-sup" Reduction Rule

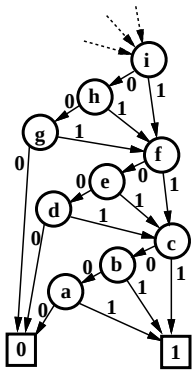


Fig. 3: 0-sup-BDD for $(a+b+c)(d+e+f)(g+h+i)\ldots$

reduction rule, the two BDDs are automatically reduced into the same form, free of irrelevant variables. 0-sup-BDDs are more effective for sparser combinations, which means that only a few objects out of many are included in each combination in the set.

The methods for manipulating 0-sup-BDDs are defined as set operations and differ slightly from those for conventional BDD manipulation. First, we generate trivial graphs and then construct more complex ones by applying basic operations such as union, intersection, and difference. We can execute these operations in a time almost proportional to the size of the graphs, just as with conventional BDDs. (see [7] for detailed algorithms.)

Using 0-sup-BDDs, we can represent and manipulate Boolean expressions efficiently. For example, when expanding the expression $(a+b+c)(e+d+f)(g+h+i)\cdots$ into a sum-of-products form, an exponential number of product terms appears for the number of variables; however, a 0-sup-BDD implicitly represents these product terms in a linear number of nodes, as shown in Fig. 3. In this graph, each path from the root to the 1-terminal corresponds to each product term in the expression. In this way, we can represent a huge number of product terms within a practical memory space.

# 3 Representation of Polynomials

Polynomial formulas are basic models in mathematics. They are often used for describing problems or procedures in various areas. Here, we represent and manipulate polynomials using 0-sup-BDDs. A method we developed for manipulating Boolean expressions is also applicable to polynomials. However, Boolean expressions are different from polynomials in the following two points:

- Boolean expressions cannot have a variable with a higher-degree.
  ($x \cdot x = x$ in Boolean algebra, not $x \cdot x = x^2$.)
- Boolean expressions cannot have a term with a coefficient.
  ($x + x = x$ in Boolean algebra, not $x + x = 2x$.)

In this section, we present a method for representing polynomials by solving those two problems.

## 3.1 Representation of Degrees

First, we show a way to deal with degrees. Here, we consider only positive integer numbers for the degrees.

The basic idea is that an integer number can be written as a sum of 2's exponential numbers by using binary coding. Namely, a variable $x^k$ can be broken down into:

$$x^k = x^{(k_1+k_2+\ldots+k_m)} = x^{k_1}x^{k_2}\ldots x^{k_m},$$

where $k_1, k_2, \ldots, k_m$ are different 2's exponential numbers. In this way, we can represent $x^k$ as a combination of $n$ items $x^1, x^2, x^4, x^8, \cdots, x^{2^{n-1}} (0 < k < 2^n)$. Such combinations can be dealt with efficiently by using 0-sup-BDDs. For example, a polynomial $x^{20} + x^{10} + x^5 + x$ can be written as $x^4x^{16} + x^2x^8 + x^1x^4 + x^1$. It can be regarded as a *set of combinations* based on the five items $x^1, x^2, x^4, x^8$, and $x^{16}$. The formula, then, can be represented by using a 0-sup-BDD, as shown in Fig. 4. In this example, we ordered $x^1$ the highest and ordered the higher degrees in lower in the graph. This ordering is convenient in calculating arithmetic operations, which is described in another section.

When dealing with more than one sort of variable, such as $x^i, y^j$, and $z^k$, we decompose them as

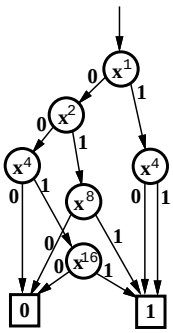



Fig. 4: $(x^4x^{16} + x^2x^8 + x^1x^4 + x^1)$

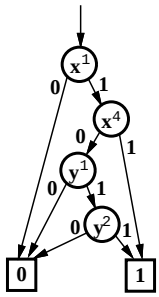



Fig. 5: $(x^1x^4 + x^1y^1y^2)$

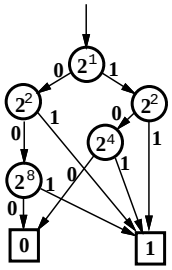



Fig. 6: $(2^8 + 2^12^4 + 2^12^2 + 2^2)$

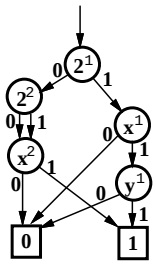



Fig. 7: $(x^2 + 2^2x^2 + 2^1x^1y^1)$

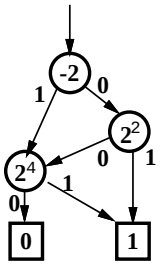



Fig. 8: $(-2 \cdot 2^4 + 2^4 + 2^2)$

$x^1, x^2, x^4, \ldots, y^1, y^2, y^4, \ldots$, and $z^1, z^2, z^4, \ldots$. Fig. 5 shows an example with two sorts of variables. Since our BDD package allows 65,535 variables, we can use more than 8,000 sorts of variables when using 8-bit coding (max 255) for degrees.

Our method features that it gives canonical forms of a polynomials, since the degrees are uniquely decomposed into the combinations based on a binary coding, and 0-sup-BDDs represent the sets of combinations uniquely. In addition, 0-sup-BDDs clearly exhibit their efficiency, for example, $x^1x^2 (= x^3)$ is represented as a combination of only $x^1$ and $x^2$, but $x^4, x^8, x^{16}, \ldots$ are not included. In 0-sup-BDDs, the nodes for irrelevant items $x^4, x^8, x^{16}, \ldots$ are automatically eliminated. Since variables with lower degrees appear more often than those with higher degrees in many cases, most of the combinations are sparse and 0-sup-BDDs are effective. In addition, when dealing with many sorts of variables, we should consider that the combination $x^1x^2$ does not include other sorts of variables, such as $y^1, y^2, y^4, \ldots$, or $z^1, z^2, z^4, \ldots$. In this case, the combinations become very sparse and 0-sup-BDDs are greatly effective.

### 3.2 Representation of Coefficients

Next, we present a way to represent coefficients. So far, we have considered only integer numbers for the coefficients.

The fundamental constant numbers "0" and "1" are represented by 0- and 1-terminal nodes in 0-sup-BDDs. Another constant number $c$ $(> 1)$ can be written as a sum of 2's exponential numbers using binary coding:

$$c = 2^{c_1} + 2^{c_2} + \ldots + 2^{c_m},$$

where $c_1, c_2, \ldots, c_m$ are different positive integer numbers. Then, regarding "2" as a symbol, just like $x, y, z$, etc., it can be represented as a polynomial of variables with degrees, which has already been discussed. Consequently, we can represent a constant number $c$ as a set of combinations from $n$ items $2^1, 2^2, 2^4, 2^8, \ldots, 2^{2^{n-1}} (0 < c < 2^{2^n})$ using 0-sup-BDDs. For example, the constant number $300 = 2^8 + 2^5 + 2^3 + 2^2$ can be written as $2^8 + 2^12^4 + 2^12^2 + 2^2$. It can be regarded as a set of combinations based on four items $2^1, 2^2, 2^4$, and $2^8$, then represented by a 0-sup-BDD, as shown in Fig. 6.

When a constant number is used as a coefficient with other variables, we can regard the symbol "2" just as one sort of variable in the formula. Figure 7 shows an example for $5x^2 + 2xy$, which is decomposed into $x^2 + 2^2x^2 + 2^1x^1y^1$.

When dealing with negative coefficients, we have to consider the coding of negative values. There are two well-known method, one of which is using 2's complement representation, and the other is using the absolute value with sign; however, both method have drawbacks. When using 2's complement, it yields many non-zero bits for small negative numbers (typically, $-1$ is "all one"), and the 0-sup. reduction rule is not effective to those non-zero bits. On the other hand, when using absolute value, the operation of addition become complicated since we have to switch the addition into subtraction for some product terms in the same formula.

To solve the above problems, we adopted another binary coding based on $(-2)^k$, namely, each bit represents $1, -2, 4, -8, 16, \ldots$. For example, $-12$ can be decomposed into $(-2)^5 + (-2)^4 + (-2)^2 = -2 \cdot 2^4 + 2^4 + 2^2$, and represented by a 0-sup-BDD as shown in Fig. 8. In this way, we can avoid to yield many non-zero bits for small negative numbers.

Two polynomials are equivalent if and only if they have the same coefficients for all corresponding terms. Since our new representation method maintains uniqueness, we can immediately check the equivalence between two polynomials after generating 0-sup-BDDs.

## 4 Algorithms for Arithmetic Operations

Polynomials can be manipulated by arithmetic operations, such as addition, subtraction, and multiplication. Based on this knowledge, we first generate 0-sup-BDDs for trivial formulas which are single variables or constants, and then apply those arithmetic operations to construct more complicated polynomials. An example is shown in Fig. 9. To generate a 0-sup-BDD for the formula $x^2+4xy$ from the arithmetic expression $x \times (x+4 \times y)$, we first generate graphs for "$x$", "$y$", and "4", then apply some arithmetic operations according to the expression. After generating 0-sup-BDDs for polynomials, we can immediately check the equivalence between two polynomials, moreover, we can easily evaluate the polynomials in terms of the length, degrees, coefficients, etc.

In this section, we present efficient algorithms for the arithmetic operations of polynomials using 0-sup-BDDs.

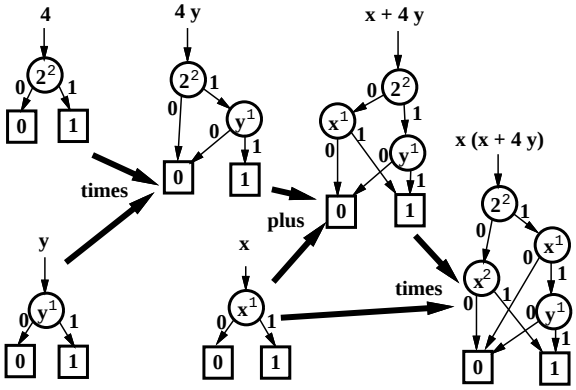


Fig. 9: Generation of 0-sup-BDDs for Arithmetic Expressions

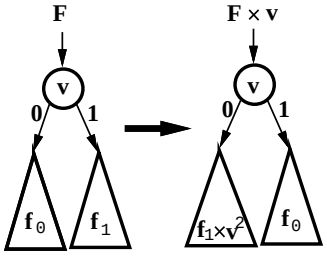


Fig. 10: 0-sup-BDDs in Multiplication by a Variable

**(Multiplication by a Variable)**
We first show an algorithm for multiplying a polynomial $F$ by a variable $v$. This operation is a basic part of other arithmetic operations. The algorithm divides $F$ into the two sub-formulas $F_1$ and $F_0$ by referring whether they contain $v$ or not. In multiplying by $v$, each product term in $F_0$ gets $v$, and each product term in $F_1$ gets $v^2$ instead of $v$. Then, $(F_1 \times v^2)$ are computed recursively. This action can be described as:
$F \times v = v \cdot F_0 \cup (F_1 \times v^2)$, where $F = v \cdot F_1 \cup F_0$
and illustrated in Fig. 10. The algorithm is executed efficiently when the variables are ordered as $x^1, x^2, x^4, x^8, \ldots$, namely, $(x^k)^2$ is always the next variable of $x^k$.

By multiplying by a special symbol $2^k$ (or $(-2)^k$), we can perform a "shift" operation for the coefficients in a formula.

**(Addition)**
If $F$ and $G$ have no common combinations, the addition $(F+G)$ can be completed by just merging them. When they contain some common combinations, we compute the following formulas:
$(F+G) = S + (C \times 2)$,
where $C = F \cap G$, $S = (F \cup G) - C$.
By repeating this process, common combinations are eventually exhausted and the procedure is completed.

We can explain the action of the algorithm using an example $F = x+z$ and $G = 3x+y$ $(= 2^1x+x+y)$. In the first execution, $C \leftarrow x$ and $S \leftarrow 2^1x+y+z$. Since $C \neq 0$, we repeat the procedure with $F = 2^1x+y+z$ $(= S)$ and $G = 2^1x$ $(= C \times 2)$. In the second execution, $C \leftarrow 2^1x$ and $S \leftarrow y+z$, and we repeat with $F = y+z$ and $G = 2^2x$. The third times, $C = 0$ and the result $2^2x+y+z$ is obtained.

When using the coding based on $(-2)^k$, the addition and subtraction can be performed as:
$(F+G) = S - (C \times (-2))$,
$(F-G) = D + (B \times (-2))$,
where $D = F \cap \overline{G}$, $B = \overline{F} \cap G$.
In this procedure, the addition and subtraction are called alternately.

**(Multiplication between Polynomials)**
Using the above two operations, we can compose an algorithm to multiply two polynomials. This algorithm is based on the divide-and-conquer idea. Suppose $v$ is the highest-ordered variable, $F$ and $G$ are then factored into two parts:

$$F = v \cdot F_1 \cup F_0, \quad G = v \cdot G_1 \cup G_0$$

Under this factorization, the product $(F \times G)$ can be written as:

$$\begin{aligned}(F \times G) &= (F_0 \times G_0) + (F_1 \times G_1) \times v^2 \\ &\quad + ((F_1 \times G_0) + (F_0 \times G_1)) \times v.\end{aligned}$$

Each sub-product term can be computed recursively. The expressions are eventually broken down into triv-

```
procedure(F × G)
{    if (F.top < G.top) return (G × F) ;
     if (G = 0) return 0 ;
     if (G = 1) return F ;
     H ← cache("F × G") ; if (H exists) return H ;
     v ← F.top ; /* the highest variable in F */
     (F_0, F_1) ← factors of F by v ;
     (G_0, G_1) ← factors of G by v ;
     H ← (F_1 × G_1) × v^2
        +((F_1 × G_0) + (F_0 × G_1)) × v + (F_0 × G_0) ;
     cache("F × G") ← H ;
     return H ;
}
```

Fig. 11: Algorithm for Multiplication between Polynomials

ial ones and the result is obtained. In the worst case, this algorithm would require exponential number of recursive calls for the number of variables; however, we can accelerate them by using a hash-based cache memory which stores the results of recent operations. By referring to the cache before each recursive call, we can avoid duplicate executions for equivalent subformulas. Consequently, the execution time depends on the size of the 0-sup-BDDs, not on the number of terms. This algorithm is given in detail in Fig. 11.

## 5 Implementation and Experiment

Based on the techniques described above, we implemented a program for manipulating polynomials. Our program is written in C++ language on a SPARC station 2 (SunOS 4.1.3, 32 MB). It can handle about 8,000 sorts of variables, up to 255 for degrees, and up to $2^{255}$ for coefficients. Our BDD package consumes about 30 bytes per node.

To evaluate our method, we constructed 0-sup-BDDs for large-scale polynomials. We first generated 0-sup-BDDs for constant numbers. In our experiment, only 15 nodes were needed to represent the number "1,000,000,000". Table 1 shows the results for $n!$. We can easily generate 0-sup-BDDs for as many as 56! within three second. (When $n = 57$, $n!$ exceeds 256 bit.)

Next, we tried to represent various kinds of polynomials, such as $x^n$, $(x+1)^n$, $\sum_{k=0}^{n} x^k$, and $\prod_{k=1}^{n}(x_k + 1)$. As shown in Tables 2 and 3, within a feasible time and space, we can generate 0-sup-BDDs for extremely large-scale polynomials, some of which consist of millions of terms. This has never been practical in conventional representation, which requires a memory space proportional to the number of terms.

Our method greatly accelerates the computation of

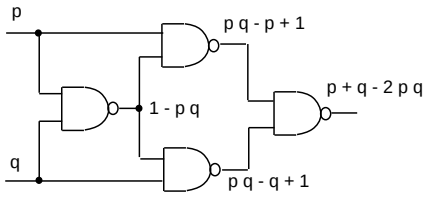



Fig. 12: Computation of Signal Probability in Logic Circuit

polynomials and enlarges the scale of applicability. It is especially effective when dealing with many sorts of variables, a feat that has been difficult with conventional methods.

## 6 Application for LSI CAD

As the polynomial calculus is a basic model in mathematics, our method is useful for various problems in LSI CAD. One good application is computing signal probability in logic circuits. As illustrated in Fig. 12, on each primary input of the circuit, we assign a variable representing the probability that the signal is '1'. Then, the probability at primary outputs and internal nets can be expressed exactly in polynomials using those probabilistic variables. These polynomials may grow large; however, our method is helpful to manipulate them efficiently. In our preliminary experiment, the polynomial for the 8 bit adder circuit grows as many as 9,841 product terms, but it can be represented by only 125 nodes of 0-sup-BDD. This technique is applicable for various kinds of statistic analysis, such as probabilistic fault simulation, estimating power consumption, and timing hazard analysis.

The formal verification of arithmetic-level description is another possible application. For example, suppose the two arithmetic expressions:

$$F_1 = (z - 2x)(6xy - 15xz + 2y^2 - 5yz),$$
$$F_2 = (3x + y)(10xz - 4xy + 2yz - 5z^2),$$

and wether $(F_1 = F_2)$ ?

By generating canonical forms of polynomials for the two expressions, the equivalence can be checked immediately. In addition, we can easily calculate the difference of the two expressions. It is useful for finding design error when they are not equivalent. In this way, our method will be utilized as a basic technique for high-level synthesis and formal verification in LSI design.

## 7 Conclusion

We have discussed a method of manipulating polynomials based on 0-sup-BDDs. We have proposed

Table 1: Result for $n!$

| n | #node | time(s) |
|---|---|---|
| 10 | 8 | 0.1 |
| 20 | 22 | 0.3 |
| 30 | 32 | 0.9 |
| 40 | 43 | 1.7 |
| 50 | 64 | 2.8 |
| 56 | 62 | 3.2 |

Table 2: Results for Polynomials (1)

| n | $x^n$ | | $n^2x^n$ | | $\sum_{k=0}^{n} x^k$ | | $\sum_{k=1}^{n}(k \times x^k)$ | | $(x+1)^n$ | |
|---|---|---|---|---|---|---|---|---|---|---|
| | #node | time(s) | #node | time | #node | time | #node | time | #node | time |
| 1 | 1 | 0.1 | 1 | 0.1 | 1 | 0.1 | 1 | 0.1 | 1 | 0.1 |
| 2 | 1 | 0.1 | 2 | 0.1 | 2 | 0.1 | 4 | 0.1 | 4 | 0.1 |
| 3 | 2 | 0.1 | 5 | 0.1 | 2 | 0.1 | 5 | 0.1 | 5 | 0.1 |
| 5 | 2 | 0.1 | 6 | 0.1 | 3 | 0.1 | 8 | 0.1 | 10 | 0.1 |
| 10 | 2 | 0.1 | 7 | 0.1 | 6 | 0.1 | 18 | 0.1 | 23 | 0.2 |
| 20 | 2 | 0.1 | 7 | 0.1 | 8 | 0.1 | 26 | 0.1 | 69 | 0.5 |
| 30 | 4 | 0.1 | 9 | 0.1 | 8 | 0.1 | 37 | 0.2 | 150 | 2.0 |
| 50 | 3 | 0.1 | 11 | 0.1 | 10 | 0.1 | 48 | 0.3 | 346 | 7.1 |
| 100 | 3 | 0.1 | 11 | 0.1 | 12 | 0.1 | 70 | 0.5 | 1,209 | 39.7 |
| 200 | 3 | 0.1 | 12 | 0.1 | 14 | 0.1 | 84 | 1.0 | 4,231 | 267.7 |
| 255 | 8 | 0.1 | 11 | 0.1 | 8 | 0.2 | 75 | 1.3 | 6,690 | 528.8 |

Table 3: Results for Polynomials (2)

| n | $\prod_{k=1}^{n}(x_k+1)$ | | | $\prod_{k=1}^{n}(x_k+k)$ | | | $\prod_{k=1}^{n}(x_k+1)^4$ | | | $\prod_{k=1}^{n}(x_k+1)^8$ | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | #term | #node | time(s) | #term | #node | time | #term | #node | time | #term | #node | time |
| 1 | 2 | 1 | 0.1 | 2 | 1 | 0.1 | 5 | 7 | 0.1 | 9 | 15 | 0.1 |
| 2 | 4 | 2 | 0.1 | 4 | 4 | 0.1 | 25 | 25 | 0.1 | 81 | 109 | 0.1 |
| 3 | 8 | 3 | 0.1 | 8 | 10 | 0.1 | 125 | 70 | 0.1 | 729 | 481 | 0.6 |
| 4 | 16 | 4 | 0.1 | 16 | 17 | 0.1 | 625 | 145 | 0.2 | 6,561 | 1,553 | 1.9 |
| 5 | 32 | 5 | 0.1 | 32 | 32 | 0.1 | 3,125 | 264 | 0.3 | 59,049 | 3,862 | 6.7 |
| 6 | 64 | 6 | 0.1 | 64 | 59 | 0.1 | 15,625 | 451 | 0.4 | 531,441 | 8,069 | 20.7 |
| 7 | 128 | 7 | 0.1 | 128 | 123 | 0.1 | 78,125 | 709 | 0.7 | 4,782,969 | 15,099 | 56.9 |
| 8 | 256 | 8 | 0.1 | 256 | 199 | 0.2 | 390,625 | 1,056 | 1.4 | 43,046,721 | 26,279 | 142.5 |
| 9 | 512 | 9 | 0.1 | 512 | 331 | 0.3 | 1,953,125 | 1,499 | 2.0 | – | (*) | – |
| 10 | 1,024 | 10 | 0.1 | 1,024 | 619 | 1.1 | 9,765,625 | 2,053 | 2.8 | – | – | – |
| 11 | 2,048 | 11 | 0.1 | 2,048 | 1,131 | 3.5 | 48,828,125 | 2,730 | 4.9 | – | – | – |
| 12 | 4,096 | 12 | 0.1 | 4,096 | 1,866 | 4.8 | 244,140,625 | 3,575 | 6.7 | – | – | – |
| 13 | 8,192 | 13 | 0.1 | 8,192 | 3,334 | 8.4 | 1,220,703,125 | 4,586 | 8.8 | – | – | – |
| 15 | 32,768 | 15 | 0.1 | 32,768 | 9,338 | 22.8 | $5^{15}$ | 7,151 | 19.7 | – | – | – |
| 20 | 1,048,576 | 20 | 0.1 | – | (*) | – | $5^{20}$ | 17,374 | 78.5 | – | – | – |

(*) Memory overflow (32 MB).

an elegant way to represent polynomials using 0-sup-BDDs, and have shown efficient algorithms for those operations. Our experimental results indicate that we can manipulate large-scale polynomials implicitly in a feasible time and space. Our representation has an important feature in that it is the canonical form of a polynomial under fixed variable ordering. As the polynomial calculus is a basic model in mathematics, our method is very useful in various areas.

In the future we will try other interesting operations in polynomials, such as differential methods, finding max/min values, solving equations, approximation, and factorization. Based on polynomial manipulation, we can extend the technique to induce more generalized models, such as rational expressions, negative or non-integer degrees, and complex (imaginary)-number coefficients.

## References


[1] R. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Trans. Comput.*, Vol. C-35, No. 8, pp. 677-691, Aug. 1986.

[2] Y. Matsunaga and M. Fujita, "Multi-level logic optimization using binary decision diagrams," *Proc. of ACM/IEEE ICCAD '89*, pp. 556-559, Nov. 1989.

[3] J. Burch, E. Clarke, K. McMillan, and D. Dill, "Sequential circuit verification using symbolic model checking," *Proc. of ACM/IEEE DAC' 90*, pp. 618-624, June 1990.

[4] O. Coudert, J. Madre and H. Fraisse, "A new viewpoint of two-level logic optimization," *Proc. of ACM/IEEE DAC '93*, pp. 625-630, June 1993.

[5] P. McGeer, J. Sanghavi, R. Brayton, and A. S.-Vincentelli: "ESPRESSO-SIGNATURE: A new exact minimizer for logic functions", *Proc. ACM/IEEE DAC '93*, pp.618-624, June 1993.

[6] B. Lin and F. Somenzi, "Minimization of symbolic relations," *Proc. of ACM/IEEE ICCAD '90*, pp. 88-91, Nov. 1990.

[7] S. Minato, "Zero-suppressed BDDs for set manipulation in combinatorial problems," *Proc. of ACM/IEEE DAC '93*, pp. 272-277, June 1993.

[8] S. Minato, "Calculation of unate cube set algebra using zero-suppressed BDDs" *Proc. of ACM/IEEE DAC '94*, June 1994 (to appear).

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