Behavioral Modeling of Analog Circuits by Dynamic Semi-Symbolic Analysis

Junjie Yang

Department of Electrical Engineering
University of California, Riverside, CA 92521, USA
jyang@ee.ucr.edu

Sheldon X.-D. Tan

Department of Electrical Engineering
University of California, Riverside, CA 92521, USA
stan@ee.ucr.edu

Abstract— *The paper presents a novel approach to behavioral modeling of analog circuits by dynamic semi-symbolic analysis, where some circuit parameters are kept as symbols and the others are given as numeric values. Our new method is based on the determinant decision diagram (DDD) representation of small-signal characteristics of linear analog circuits. The basic idea is to dynamically reorder DDD vertices such that all the DDD vertices corresponding to symbolic parameters are separated from DDD vertices for numerical parameters. In this way, DDD sizes of symbolic portion of DDD can be significantly reduced by suppressing numerical DDD nodes. Our new approach is different from the existing MTDDD based semi-symbolic analysis method where reordering is done before DDD is constructed and DDD-based graph operations are still valid in the new method. The proposed dynamic ordering algorithm, which is based on swap of adjacent variables, also improves the existing DDD-based vertex sifting algorithm as no special sign rule is required after DDD vertices are swapped. Experimental results have demonstrated that the proposed dynamic semi-symbolic method leads to up to 30% symbolic DDD node reduction compared MTDDD method on real analog circuits and can be performed very efficiently.*

I. Introduction

Symbolic analysis is devoted to the analysis of integrated circuits in which part or all the circuit parameters and the complex frequency variable are represented by symbols. As illustrated in [1], simple yet accurate symbolic expressions can also be interpretable by analog designers to gain insight into circuit behavior, performance and stability, and are important for many applications in circuit design such as transistor sizing and optimization, topology selection, sensitivity analysis [2]. But its applications have traditionally suffered from the exponential growth of the complexity of the symbolic results with the circuit size. This has motivated many approximate techniques [3]. But the approximated expressions will lose some information which are crucial in some applications.

From analog circuit design standpoints, most of the time, only some circuit parameters are of interest for a specific circuit design task. This leads to the semi-symbolic analysis where only a small portion of circuit parameters are kept as symbols and the rest of parameters are given as numerical values. This semi-symbolic analysis brings the benefits of reduced symbolic expressions at no cost of accuracy loss.

Semi-symbolic analysis was studied in the past by numerical interpolation method [4], by parameter extraction [5], and by using multi-terminal DDD concepts [6]. But existing approaches either have very restrictive usage or suffer circuit-size limitation problems. Recently, a multi-terminal DDD (MTDDD) graph was proposed to represent the semi-symbolic expressions [6]. The idea is to order the complex DDD nodes before DDD constructions such that the DDD nodes, whose matrix elements contain symbolic circuit parameters, are positioned above all DDD nodes whose matrix elements contain only numerical circuit parameters when the complex DDD graph is constructed. The *s*-expanded MTDDDs are then constructed in a bottom up fashion. But the resulting MTDDDs do not hold many DDD properties and they are difficult to perform many DDD graph operations, which are crucial to many symbolic analysis tasks, on the MTDDD graphs. Worse, some numerical MTDDD nodes may appear above symbolic MTDDD nodes in the resulting MTDDDs. As a result, size of MTDDDs is not adequately reduced and benefits of MTDDDs are not fully explored. For example, in Fig. 1, numerical node **2** will be the parent node of symbolic node sC_1 in the resulting MTDDD graph. As a result, the MTDDD may still be large as those numerical DDD nodes can be suppressed.

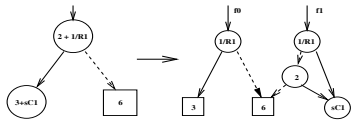

Fig. 1. A MTDDD graph which has numerical DDD nodes.

In this paper, we propose a new approach to dynamic semi-symbolic analysis of analog circuits. The new method also starts with exact symbolic representations of small-signal circuit characteristics of linear analog circuit via determinant decision diagrams [7–9]. The new method does not explicitly construct the MTDDDs, instead it first constructs normal *s*-expanded DDDs from complex DDDs. Then a dynamic variable ordering procedure is performed to move all the symbolic *s*-expanded DDD nodes to the top of the DDD graph. After this, all the bottom numerical DDD nodes can be suppressed implicitly to reduce overall DDD sizes. The resulting *s*-expanded DDDs are constructed such that the MTDDD graphs are implicitly embedded in the top portion of the *s*-expanded DDD graphs. The new approach offers several advantages over MTDDD-based method. First, all the DDD-based graph operations can still be performed on the resulting *s*-expanded DDD graphs as not MTDDD graphs are constructed. Second, it will result in smaller size of DDD graphs. Third, with the freedom of moving nodes to different positions of a DDD graph, the new method is able to change the set of symbolic parameters dynamically without re-performing the symbolic analysis each time. We will show how dynamic variable ordering is performed on the DDD graphs via adjacent variable order swap, which does not require any new sign rule for the moved DDD vertices. This is in contrast to the DDD size optimization method based on a vertex swap method in [10], where a special sign rule was developed for the swapped DDD vertices and the proposed sign rule has significant impacts on the effects of DDD minimization.

II. DDDs and DDD Implementations

In this section, we first review the notion of determinant decision diagrams. Then we present some implementation details that are important for the new dynamic variable ordering algorithm.

Determinant Decision Diagrams [7] are compact and canonical graph-based representation of determinants. It represents a determinant that has an associated variable x_i and points to two other nodes (cofactors of the determinant) in the graph. The node D is written as a tuple $(I(x_i), D_{x_i}, D_{\overline{x}_i})$ where x_i is called the *top* variable of the determinant D, D_{x_i}, $D_{\overline{x}_i}$ are the cofactor and remainder of D with respect to x_i respectively. $I(x_i)$ is the index of x_i. For convenience, D_{x_i}, $D_{\overline{x}_i}$ are also written as X_1 and X_0 in the sequel. DDD is a ordered graph where an index is assigned to each variable and the

variable must appear in descending order in terms of their indices along each path in the DDD. The index of variable is also called *level* of the variable.

Following a similar implementation of BDDs, a global hash table, called *unique* table, allows a node of DDD node $(I(x_i), X_1, X_0)$ to be found in a constant time. The hash table typically is implemented by an array of hash-trees – *unique[ ]* where all level-j DDD nodes are stored in *unique[j]* in a tree structure based on their hash values of tuple $(I(x_j), X_1, X_0)$. The advantage of such two-level hashing implementation is that we can access all the nodes at level j without walking the entire unique table. This is important for the dynamic variable ordering as it will become clear later.

III. Dynamic Variable Ordering For DDDs

It has been shown that the order swap of two adjacent variables in an OBDD affects only the BDD nodes at the two levels and all other nodes remain unchanged [11–13]. It turns out that this is also true for DDD graph as each DDD node represent a determinant.

As we mentioned before that we can access all the nodes at level i directly. This makes the variable swap very memory efficient as we don't need to touch nodes at other levels. Suppose we want to swap the variable x at level i and y at level j with $i > j$. i.e. $I(x) = i$ and $I(y) = j$. After the swap, the indices of the two variables will become: $I_a(x) = j$ and $I_a(y) = i$. Suffix a means the index after the order swap.

The basic idea behind variable swap is that we maintain an identical determinant represented by each node even though indices of the nodes are changed. Meanwhile, we also make sure that descending variable order is enforced after swap. The major difference between BDD and DDD is that we need to determine the signs of swapped DDD nodes, It turns out that the sign rule is still valid. We have the following lemma without proof.

Lemma 1 *The sign rule proposed in [7] is still applicable to the swaped DDD nodes after the swap of their variable orders.*

To perform the swap operation, we first consider all the DDD nodes at level i. Those nodes can be further classified into two types: (1) node x in which the top variables of both its two child nodes X_1 and X_0 are not y and (2) nodes otherwise. For the first type nodes, the reordering procedure is shown in Fig. 2. The new node DDD node y with its 0-edge pointing to x and 1-edge pointing to 0 terminal is created first in step (b). Since for a DDD graph, node pointing to 1 terminal has to suppressed (zero suppression rule), we end up with a DDD graph shown in step (c), the index of x becomes j after the swap, so the whole operation is equivalent to overwriting node x with a new index $I_a(x) = j$.

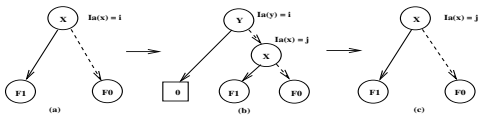

Fig. 2. Variable swap for the first type of the nodes at level i.

For the second type nodes, we follow the same method in [13] as follows: For each node $x = (I(x), X_1, X_0)$ at level i, we create a new node, $(I_a(y), (I_a(x), F_{11}, F_{01}), (I_a(x), F_{01}, F_{00}))$, to overwrite it, where X_{11} and X_{10} which are the cofactor and remainder of X_1 with respect to y respectively, and X_{01} and X_{00} are the cofactor and remainder of X_0 with respect to y respectively. X_{11} and X_{10} and X_{01} and X_{00} can be computed very easily by using COFACTOR(D,P) and REMAINDER(D,P) operations [7] and they take a constant time in this particular case as x and y are adjacent variables.

For each node y at level j, we can simply overwrite its index with $I_a(y) = i$ after all the level i nodes are processed. Note that after we change the index of a node, we need to put it into a new position in the unique table. Let i and j be the adjacent indices with $i > j$. The pseudo algorithm of the adjacent variable swap method is shown in Fig. 3.

```
SWAPADJACENTVARIABLEORDER(j,i)
01  for each node x = (i,X1,X0) in unique[i] do // process type 1 nodes
02    if(I(X1) is not j and I(X0) is not j)
03      Overwrite the node with (j,X1,X0) and put it into a hash tree Hj
04  for each node x in unique[i] do // process type 1 nodes
05    if(I(X1) = j or I(X0) = j)
06      X11 = COFACTOR(X1, j)
07      X10 = REMAINDER(X1, j)
08      X01 = COFACTOR(X0, j)
09      X00 = REMAINDER(X0, j)
10      Create nodes F1 = (j,X11,X01) and F0 = (j,X10,X00)
11      Check and put new node (j,F1,F0) into the Hj
12  for each node y = (j,Y1,Y0) in unique[j] do
13    Overwrite the node with (i,Y1,Y0) and put it into the hash tree Hi
14  unique[j] = Hj
15  unique[i] = Hi
```

Fig. 3. Dynamic adjacent variable swap algorithm for DDD graphs

It is easily to see that time complexity of the algorithm is linearly proportional to the number of DDD nodes in level i and level j. As a result, garbage collection is also performed to reduce the unreferenced nodes at level i and j before the swap operations.

IV. Semi-Symbolic Analysis via Dynamic Variable Ordering

With the efficient adjacent variable swap algorithm, semi-symbolic analysis essentially boils down to moving all the symbolic DDD nodes to the top of the DDD graph and suppress all unused numerical DDD nodes if necessary. The concept is best illustrated using a simple RC filter circuit shown in Fig. 4. Its system equations can be written as

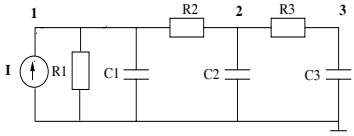

Fig. 4. A simple RC circuit.

$$\begin{bmatrix} \frac{1}{R_1}+s\mathbf{C_1}+\frac{1}{R_2} & -\frac{1}{R_2} & 0 \\ -\frac{1}{R_2} & \frac{1}{R_2}+sC_2+\frac{1}{R_3} & -\frac{1}{R_3} \\ 0 & -\frac{1}{R_3} & \frac{1}{R_3}+s\mathbf{C_3} \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} I \\ 0 \\ 0 \end{bmatrix}$$

Let C_1 and C_3 be two symbolic parameters in the circuit. Matrix entries $\frac{1}{R_1}+sC_1+\frac{1}{R_2}$ and $\frac{1}{R_3}+sC_3$ will be assigned indices larger than the indices of all other entries to make them appear on the top of the corresponding DDD graph. Let T be the 3×3 system matrix and we are interested in the following transfer function

$$H(s) = \frac{V_3(s)}{I(s)} = \frac{(-1)^{1+3} det(T_{13})}{det(T)}, \qquad (1)$$

where T_{13} is the matrix obtained by removing row 1 and column 3 from T. The resulting determinant of the system matrix, its cofactor T_{13}, and their DDD representations are shown in Fig. 5, where each non-zero element is designated by a symbol and is assigned a unique index in parentheses. The index of each symbol is also marked along each DDD node in the resulting DDD graph. It can be seen that symbolic nodes A and G appear above all the other numerical DDD nodes.

Once complex DDDs are obtained, s-expanded DDDs are can be computed very efficiently [8]. Consider again the circuit in Fig. 4 and its system determinant. Let us introduce a unique symbol for each circuit parameter in its admittance form. Specifically, we introduce $a = \frac{1}{R_1}$, $b = f = \frac{1}{R_2}$, $d = e = -\frac{1}{R_2}$, $g = k = \frac{1}{R_3}$, $i = j = -\frac{1}{R_3}$,

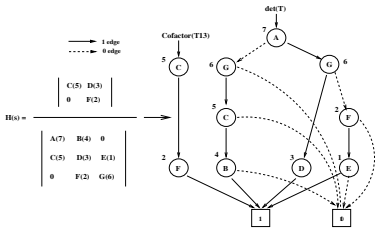

Fig. 5. A complex pre-ordered DDD for the transfer function

$C_1 = c, h = C_2, l = C_3$. Then the circuit matrix can be rewritten as

$$\begin{bmatrix} a+b+\mathbf{c}s & d & 0 \\ e & f+g+hs & i \\ 0 & j & k+\mathbf{l}s \end{bmatrix}$$

The corresponding s-expanded DDDs are shown in Fig. 6. If a circuit parameter is symbolic, its corresponding DDD nodes are symbolic, otherwise they are numerical DDD nodes. For example, c represents a symbolic term C_1, so its DDD nodes are symbolic. All the symbolic DDD nodes are represented by a shaded circle in Fig. 6. We notice that numerical DDD nodes a, b still appear above symbolic DDD nodes c and l as complex DDD node of index A is above complex node of index D and s-expanded DDD construction process does not change the relative order of the resulting s-expanded DDD nodes.

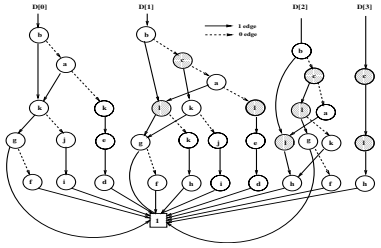

Fig. 6. Semi-symbolic s-expanded DDDs for $det(T)$

We then perform the dynamic variable swap to move all the symbolic DDD nodes to the top of DDD graph such that all symbolic nodes are above numerical DDD nodes. The process is done for each index whose nodes are symbolic. Each time, we swap two adjacent variables at a time until the symbolic index reach to a position where no other numerical indices are larger than it. After this, we process next symbolic index until all the symbolic indices are above numerical indices. The resulting re-ordered s-expanded DDD graphs are shown in Fig. 7.

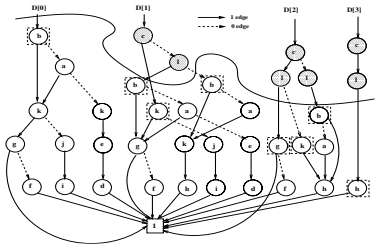

Fig. 7. Semi-symbolic s-expanded DDDs for $det(T)$ after dynamic variable ordering

After the dynamic variable ordering, numerical and symbolic DDD nodes are completely separated. A line is draw between the two portions of the DDD nodes in Fig. 7. If we only keep all symbolic nodes and those nodes whose parent is a symbolic node, designated by a dotted box, we end up with a MTDDD graph where the left numeral nodes are the numerical terminals in the MTDDD. The numerical values at those embedded numerical terminals can be easily obtained by a DFS traversal of the DDD tree rooted at those nodes and all the other numerical DDD nodes can be implicitly removed by garbage collection. Moreover many DDD graph operations can still be applied if we still keep the numerical nodes. For example, we can find the dominant terms in the semi-symbolic DDD by using the fastest incremental shortest path algorithm [14]. The difference is that once we find one shortest path, we need to subtract all the paths which go through the numerical terminal on the shortest path.

Another advantage of the new algorithm over MTDDD method is that we can dynamically change the symbolic circuit parameter sets without reconstructing the DDDs and sDDDs. As a result, the new algorithm is very amenable to analog circuit design and optimization applications where such changes are encountered very frequently.

V. Experimental Results

The proposed approach has been implemented and tested on a number of practical analog circuits ranging from more regularly structured ladder circuits to less regularly structured such as *Cascode* and $\mu A741$ amplifiers. The algorithms described in [7, 8] are used to construct complex DDDs and s-expanded DDDs for a transfer function for each circuit. A Linux system with dual 500Mhz Intel Celeron CPUs and 300M memory is used for all the following experiments.

Table II summarizes statistics of semi-symbolic s-expanded symbolic expressions at different stages of operations. In Table II, column 1, 2 and 3 list for each circuit, respectively, its name (*Circuit*), the number of nodes (*#nodes*), and the number of symbolic branches selected and total number of branches of the circuit (*Symb #bch/#bch*). Columns 4 to 5 show, respectively, the number of s-expanded DDD nodes ($|sDDD|$ *before DVO*) before dynamic variable ordering (DVO) for a transfer function, and the number of s-expanded DDD nodes after DVO ($|sDDD|$ *after DVO*). Columns 6 to 7 present, respectively, the number of symbolic DDD nodes (*Symb* $|sDDD|$), and number of embedded numerical terminals (*Num #terminal*). Column 8, ($|MTDDD|$), shows the number of MTDDD nodes, which consists of both numerical and symbolic DDD nodes. Those numerical nodes are parent nodes of some symbolic nodes. They are obtained by only pre-ordering complex DDD variables without dynamic variable ordering. The last column in the table presents the DDD size reduction percentage of symbolic $|sDDD|$ over $|MTDDD|$.

From the table we can observe sizes of symbolic sDDD are smaller than sizes of MTDDDs for all circuits except for *rclad21*. The reduction percentage ranges from 10% to 30%. This is significantly improvement especially for real analog circuits like *Cascode* and $\mu A741$ Opamps. For *rclad21* circuit, the reason that the size of MTDDDs is smaller than that of symbolic sDDDs is that it is a ladder circuit and the resulting sDDD has a variable ordering close to optimal ordering, dynamic variable ordering will increase the DDD size which offsets the DDD size reduction provided by bringing symbolic DDD nodes together. This also reflects the fact that after dynamic variable ordering, the total sDDD sizes get increased for all the circuits. But the increase will not be significant if the number of symbolic branches are not large. Also we notice that if the number of symbolic branches become larger (more than 10), the benefit of semi-symbolic analysis will begin to lose as the symbolic DDD portion itself will become significant large.

Table I shows the CPU time for constructing and evaluating the semi-symbolic DDD for the same transfer function of each circuit. For comparison reason, we also list the CPU time for constructing complex DDDs when the best variable ordering is used [7] in Column 2 (*cDDD (w/o preorder)*). Column 3 lists the CPU time of complex DDD construction when pre-ordering is used to put all the symbolic complex DDDs to the top of complex DDD tree (*cDDD*

(w preorder)). It appears that the CPU time does increase with pre-ordering, but it become less significantly for large less-structured circuits like $\mu A741$. Therefore, the CPU time for complex DDD construction is still very fast in the presence of a few symbolic branches.

TABLE I
CPU TIME STATISTICS ON SEMI-SYMBOLIC DDD OPERATIONS.

Circuit	cDDD (w/o preorder)	cDDD (w preorder)	*s*DDD	DVO	Eval of *s*DDD	Eval of Symb DDD
rlc6	0.02	0.03	0.06	0.02	0.10	0.02
rlc14	0.16	1.48	6.65	15.09	34.8	1.4
Cascode	0.32	0.95	2.49	4.88	3.50	0.04
$\mu A741$	1.44	1.59	5.76	0.21	15.22	0.06
rctreeA	0.10	0.14	0.24	0.57	1.85	0.24
rctreeB	0.17	0.31	0.74	0.24	6.49	0.41
bigtst	0.34	0.38	0.43	0.19	2.27	0.14
rclad21	0.02	0.03	0.03	< 0.01	0.05	0.01

Column 4 to 5 gives the CPU time for constructing the *s*-expanded DDD and for performing the DVO. Both CPU times are essentially proportional to the size of *s*-DDDs and can be performed very efficiently. The last two columns present the CPU time for taking the numerical evaluation of all the coefficients for the whole *s*-expanded DDD graph (*Eval of sDDD*) and for only the symbolic portion of the *s*-expanded DDD graph (*Eval of Symb DDD*) after DVO. So the benefits of evaluating only symbolic sDDD are obvious.

VI. CONCLUSIONS

This paper proposes a novel approach to semi-symbolic analysis of large analog circuits based on DDD graphs. We show how DDD nodes in adjacent variables in DDD graphs can be swapped in order, which in turn can be used to reduce the DDD presentation of a linearized analog circuits via dynamic variable ordering. The new dynamic variable ordering method does not require special sign rule for order-swapped DDD vertices compared with the existing method and is able to dynamically change symbolic circuit parameter sets. The new method also preserve all the DDD graph operation capabilities compared to MTDDD method. Experimental results have demonstrated that the proposed semi-symbolic method leads to up to 30% symbolic DDD node reduction compared with MTDDD method on real analog circuits and can be performed very efficiently.

TABLE II
STATISTICS ON SEMI-SYMBOLIC *s*-EXPANDED DDDS

Circuit	#nodes	Symb #bch/#bch	$\|sDDD\|$ before DVO	$\|sDDD\|$ after DVO	Symb $\|sDDD\|$	Num #terminal	$\|MTDDD\|$	Imprv(%)
rlc6	13	8/19	2466	2697	639	260	700	8.7%
rlc14	29	8/45	2.84×10^5	3.51×10^5	13198	11720	18174	27.4%
Cascode	14	5/78	74778	81577	942	813	1311	28.2%
$\mu A741$	23	5/93	2.14×10^5	2.15×10^5	693	441	1098	36.9%
rctreeA	40	8/80	13027	13211	1629	170	1785	8.7%
rctreeB	53	8/106	36407	36579	2246	238	2450	8.3%
bigtst	32	5/51	20082	20514	1230	474	1396	11.9%
rclad21	22	8/41	630	660	184	21	173	-6.4%

REFERENCES

[1] G. Gielen and W. Sansen, *Symbolic Analysis for Automated Design of Analog Integrated Circuits*. Kluwer Academic Publishers, 1991.

[2] G. Gielen, P. Wambacq, and W. Sansen, "Symbolic analysis methods and applications for analog circuits: A tutorial overview," *Proc. of IEEE*, vol. 82, pp. 287–304, Feb. 1994.

[3] F. V. Fernández, A. Rodríguez-Vázquez, J. L. Hertas, and G. Gielen, *Symbolic Analysis Techniques: Application to Analog Design Automation*. IEEE Press, 1998.

[4] K. Singhal and J. Vlach, "Generation of immittance functions in symbolic form for lumped distributed active networks," *IEEE Trans. on Circuits and Systems I: Fundamental Theory and Applications*, vol. CAS-21, pp. 39–45, 1974.

[5] P. Sannuti and N. N. Puri, "Symbolic network analysis—an algebraic formulation," *IEEE Trans. on Circuits and Systems I: Fundamental Theory and Applications*, vol. 27, pp. 679–687, Aug. 1980.

[6] T. Pi and C.-J. Shi, "Multi-terminal determinant decision diagrams: a new approach to semi-symbolic analysis of analog integrated circuits," in *Proc. Design Automation Conf. (DAC)*, pp. 19–22, June 2000.

[7] C.-J. Shi and X.-D. Tan, "Canonical symbolic analysis of large analog circuits with determinant decision diagrams," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 19, pp. 1–18, Jan. 2000.

[8] C.-J. Shi and X.-D. Tan, "Compact representation and efficient generation of s-expanded symbolic network functions for computer-aided analog circuit design," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, pp. 813–827, April 2001.

[9] W. Verhaegen and G. Gielen, "Efficient ddd-based symbolic analysis of large linear analog circuits," in *Proc. Design Automation Conf. (DAC)*, pp. 139–144, June 2001.

[10] A. Manthe and C.-J. R. Shi, "Lower bound based ddd minimization for efficient symbolic circuit analysis," in *Proc. IEEE Int. Conf. on Computer Design (ICCD)*, pp. 374–379, Nov. 2001.

[11] M. Fujita, Y. Matsunaga, and T. Kauda, "On variable ordering of binary decision diagrams for the application of multi-level logic synthesis," in *Proc. European Design Automation Conf.*, pp. 40–54, March 1991.

[12] N. Ishiura, H. Sawada, and S. Yajima, "Minimization of binary decision diagrams based on exchanges of variables," in *Proc. Int. Conf. on Computer Aided Design (ICCAD)*, pp. 472–475, Nov. 1991.

[13] R. Rudell, "Dynamic variable ordering for ordered binary decision diagrams," in *Proc. Design Automation Conf. (DAC)*, pp. 42–47, June 1993.

[14] S. X.-D. Tan and C.-J. Shi, "Efficient ddd-based term generation algorithm for analog circuit behavioral modeling," in *Proc. Asia South Pacific Design Automation Conf. (ASPDAC)*, pp. 789–794, Jan. 2003.