Canonical Symbolic Analysis of Large Analog Circuits with Determinant Decision Diagrams

C.-J. Richard Shi, *Senior Member, IEEE* and Xiang-Dong Tan, *Member, IEEE*

***Abstract*—Symbolic analysis has many applications in the design of analog circuits. Existing approaches rely on two forms of symbolic-expression representation: expanded sum-of-product form and arbitrarily nested form. Expanded form suffers the problem that the number of product terms grows exponentially with the size of a circuit. Nested form is neither canonical nor amenable to symbolic manipulation. In this paper, we present a new approach to exact and canonical symbolic analysis by exploiting the *sparsity* and *sharing* of product terms. It consists of representing the symbolic determinant of a circuit matrix by a graph—called a determinant decision diagram (DDD)—and performing symbolic analysis by graph manipulations. We show that DDD construction, as well as many symbolic analysis algorithms, takes time almost linear in the number of DDD vertices. We describe an efficient DDD-vertex-ordering heuristic and prove that it is optimum for ladder-structured circuits. For practical analog circuits, the numbers of DDD vertices are several orders of magnitude less than the numbers of product terms. The algorithms have been implemented and compared respectively to symbolic analyzers *ISAAC* and *Maple-V* in generating the expanded sum-of-product expressions, and *SCAPP* in generating the nested sequences of expressions.**

***Index Terms*—Analog symbolic analysis, circuit simulation, determinant decision diagrams (DDD's), symbolic matrix determinant, zero-suppressed binary decision diagram (ZBDD).**

I. Introduction

SYMBOLIC analysis calculates the behavior or characteristic of a circuit in terms of symbolic parameters. In contrast to numerical simulators such as *SPICE* [32] that only provide numerical results, symbolic simulators can explicitly express which circuit parameters determine the circuit behavior. They can offer more advantages than numerical simulators in many applications such as optimum topology selection, design space exploration, behavioral model generation, and fault detection; these have been summarized and illustrated by a recent survey paper of Gielen *et al.* [22].

Despite its advantages, symbolic analysis has not been widely used by analog designers and was once judged as completely inefficient [28]. The root of the difficulty is apparent: the number of product terms in a symbolic expression may increase exponentially with the size of a circuit. For example, for a BiCMOS amplifier that has about 15 nodes and 25 devices (transistors, diodes, resistors, and capacitors), the determinant of the circuit matrix contains more than 10^{11} product terms [47]. Any manipulation and evaluation of sum-of-product-based symbolic expressions will require CPU time, at best, linear in the number of terms and, therefore, have both time and space complexities exponential in the size of a circuit.

To cope with large analog circuits, modern symbolic analyzers[1] rely on two techniques—hierarchical decomposition and symbolic simplification. Hierarchical decomposition generates symbolic expressions in the nested instead of expanded form [23], [24], [40]. Symbolic simplification discards those insignificant terms based on the relative magnitudes of symbolic parameters and the frequency defined at some nominal design points or over some ranges. It can be performed before/during the generation of symbolic terms [25], [37], [45], [51] or after the generation [17], [21], [47]. Exploitation of these techniques has enabled the use of symbolic simulators in several university research projects [9], [19], [21], [50]; however, both techniques have some major deficiencies. Symbolic manipulation (other than numerical evaluation) of a nested expression usually requires complicated and time-consuming procedures; e.g., sensitivity calculation in [27] and lazy approximation in [37]. On the other hand, simplified expressions only have a sufficient accuracy at some points or frequency ranges. Even worse, simplification often loses certain information, such as sensitivity with respect to parasitics, which is crucial for circuit optimization and testability analysis.

In this paper, we present a new approach to exact symbolic analysis, which is capable of analyzing analog integrated circuits substantially larger than those previously handled. Our approach is based on two observations concerning symbolic analysis of large analog circuits: 1) the circuit matrix is sparse and 2) a symbolic expression often shares many subexpressions. Under the assumption that all the matrix elements are distinct, each product term can be viewed as a subset of all the symbolic parameters. Therefore, we adapt a special data structure called ZBDD's[2] introduced originally for representing sparse subset systems [29]. This leads to a new graph representation of symbolic determinants, called DDD's. This representation has several advantages over both the expanded and arbitrarily nested

Manuscript received June 4, 1997; revised April 10, 1999. This work was supported in part by the U.S. Defense Advanced Research Projects Agency (DARPA) under Grant F33615-96-1-5601, in part by United States Air Force, Wright Laboratory, Manufacturing Technology Directorate, and in part by Conexant Systems. A preliminary version of this paper was presented at the IEEE/ACM Int. Conf. Computer-Aided Design, San Jose, CA, November 9–13, 1997 [38]. This paper was recommended by Associate Editor M. Fujita.

C.-J. R. Shi is with the Department of Electrical Engineering, University of Washington, Seattle, WA 98195 USA (e-mail: cjshi@ee.washington.edu).

X.-D. Tan is with the Department of Electrical Engineering, University of Washington, Seattle, WA 98195 USA.

Publisher Item Identifier S 0278-0070(00)01381-6.

[1]Some are surveyed in a paper by F. V. Fernández and A. Rodríguez-Vázquez [19].

[2]We remark that ZBDD's is a variant of binary decision diagrams (BDD's) introduced by Akers [1] and popularized by Bryant [4]. Our work is inspired by the success of BDD's as an enabling technology for industrial use of symbolic analysis and formal verification in digital logic design [5].

forms of a symbolic expression. First, similar to the nested form, our representation is compact for a large class of analog circuits. A ladder-structured network can be represented by a diagram with the number of vertices (called its *size*) equal to the number of symbolic parameters. As indicated by our experiments, the size of a DDD is usually dramatically smaller than the number of product terms. For example, 5.71×10^{20} terms can be represented by a diagram with 398 vertices. Second, similar to the expanded form, our representation is canonical; i.e., every determinant has a **unique** DDD representation. The representation canonicity facilitates efficient symbolic analysis and may provide a potential tool to formally verify analog circuits. Finally, derivation, manipulation and evaluation of the DDD representations of symbolic determinants have time complexity proportional to the DDD sizes.

This paper is organized as follows: Section II introduces the background and basic notation for the rest of the paper. Section III presents the notion of determinant decision diagrams as an application of ZBDD's to represent symbolic determinants. Section IV describes an effective heuristic for ordering DDD vertices so that the resulting DDD has a smallest or near-smallest size. DDD-based algorithms for symbolic analysis and applications are described in Section V. Experimental results are presented in Section VI. The proposed approach is compared to some related work in Section VII. Section VIII concludes the paper.

II. Notations and Problem Statement

In this section, we introduce some basic notation and concepts that will be used in the rest of the paper. Since these come from several different research areas, an attempt has been made to choose a self-consistent set of notations.

A. Subset Systems and ZBDD's

Let V be a *set* of elements. The number of elements in V is called the *cardinality* of V, denoted by $|V|$. The set of all *subsets* of V is called the *power set* of V, denoted by 2^V. A subset X of the power set, written as $X \subseteq 2^V$, is called a *subset system* of V.

A subset system X of V can be decomposed with respect to an element v in V into two unique subset systems, X_v and $X_{\overline{v}}$, where X_v is the set of subsets of V belonging to X that contain v, from which v has been removed, and $X_{\overline{v}}$ is the set of subsets of V belonging to X that do not contain v. For instance, let $X = \{\{v_1, v_2\}, \{v_1, v_3, v_5\}, \{v_3, v_4, v_5\}\}$. Then we have $X_{v_1} = \{\{v_2\}, \{v_3, v_5\}\}$, and $X_{\overline{v}_1} = \{\{v_3, v_4, v_5\}\}$. This decomposition can be represented graphically by a decision *vertex*. It is labeled by v_1, and represents the subset system X. As illustrated in Fig. 1(a), the vertex has two outgoing edges: one points to X_{v_1} (called *1-edge*), and the other to $X_{\overline{v}_1}$ (called *0-edge*). We say that the edges are *originated* from the vertex. If X is recursively decomposed with respect to all the elements of V, one obtains a binary decision tree whose leaves are {{}} and {}, respectively. For convenience, we denote leaf {{}} by the 1-terminal, and {} by the 0-terminal.

When a subset system X is decomposed with respect to an element that does not appear in X, then its 1-edge points to the 0-terminal. This is illustrated in Fig. 1(b) for decomposing $X = \{\{v_3, v_4, v_5\}\}$ with respect to v_2. To make the diagram compact, Minato suggested the following *zero-suppression* rule for representing sets of *sparse* subsets: eliminate all the vertices whose 1-edges point to the 0-terminal vertex and use the subgraphs of the 0-edges, as shown in Fig. 1(b) [29]. A ZBDD is such a zero-suppressed graph with the following two rules due to Bryant [4]: *ordered*—all elements of V, if one appears, will appear in a fixed order in all the paths of the graph; and *shared*—all equivalent subgraphs are shared. ZBDD's are a canonical representation of subset systems; i.e., every subset system has a unique ZBDD representation under a given vertex ordering. For example, Fig. 1(a) is a unique ZBDD representation for the subset system $\{\{v_1, v_2\}, \{v_1, v_3, v_5\}, \{v_3, v_4, v_5\}\}$ with respect to ordering v_1, v_2, v_4, v_3 and v_5. For convenience, every nonterminal vertex is indexed by an integer number greater than those of its descendant vertices [29]. The process of assigning indexes to the nonterminal vertices is called *vertex ordering*.

A path from a nonterminal vertex to the 1-terminal is called *1-path*. It defines a subset of V. The subset consists of all the elements of V from which the 1-edges in the 1-path originate. A

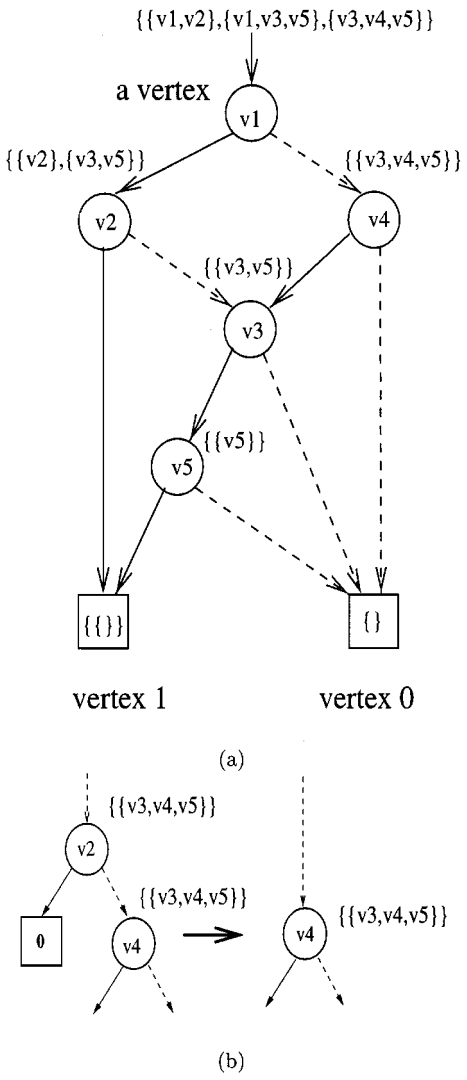


Fig. 1. (a) A ZBDD example and (b) an illustration of the zero-suppression rule.

1-path from the root is a *rooted* 1-path. The number of vertices in a ZBDD is called its *size*.

B. Matrix, Determinant, and Cofactors

Let $e = \{1, \cdots, n\}$ be a set of integers. Let A denote a set of m elements, called *symbolic parameters* or simply *symbols*, $\{a_1, \cdots, a_m\}$, where $1 \le m \le n^2$ and each symbol is labeled by a unique pair (r, c), where $r \in e$ and $c \in e$. Often, we write A as an $n \times n$ (square) *matrix*, denoted by $\boldsymbol{A}$, and use $a_{r,c}$ to denote the element of matrix $\boldsymbol{A}$ at row r and column c. We sometimes use $r(a)$ and $c(a)$ to denote, respectively, the row and column indexes of element a

$$\boldsymbol{A} = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{pmatrix}.$$

If $m = n^2$ the matrix is said to be *full*. If $m \ll n^2$ the matrix is said to be *sparse*. The *determinant* of $\boldsymbol{A}$, denoted by $\det(\boldsymbol{A})$, is defined by

$$\begin{vmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{vmatrix} = \sum_{j_1 \ne j_2 \ne \cdots \ne j_n} (-1)^p \cdot a_{1,j_1} \cdot a_{2,j_2} \cdots a_{n,j_n}. \quad (1)$$

Here

$(j_1, j_2, \cdots, j_n)$	permutation of e;
p	number of permutations needed to make the sequence $(j_1, j_2, \cdots, j_n)$ monotonically increasing.

The right-hand side of (1) is a symbolic expression of $\det(\boldsymbol{A})$ in the *expanded* form, more precisely, the *sum-of-product* form, where each *term* is an algebraic product of n symbolic parameters. We note that each symbol can be assigned a real or complex value for analog circuit simulation.

Let p, $p \subseteq e$, and q, $q \subseteq e$ such that $|p| = |q|$. The square matrix obtained from the matrix $\boldsymbol{A}$ by deleting those rows not in p and columns not in q forms a *submatrix* of $\boldsymbol{A}$, and is represented by $\boldsymbol{A}(p, q)$. It has dimension $|p|$ by $|q|$.

Let $a_{r,c}$ be the element of $\boldsymbol{A}$ at row r and column c. Let $\boldsymbol{A}_{a_{r,c}}$ be the $(n-1) \times (n-1)$-matrix obtained from the matrix $\boldsymbol{A}$ by deleting row r and column c, and let $\boldsymbol{A}_{\overline{a}_{r,c}}$ be the $n \times n$-matrix obtained from $\boldsymbol{A}$ by setting $a_{r,c} = 0$. Then, the determinant of matrix $\boldsymbol{A}$ can be *expanded* in a way similar to Shannon expansion for Boolean functions [8]

$$\det(\boldsymbol{A}) = a_{r,c}(-1)^{r+c} \det(\boldsymbol{A}_{a_{r,c}}) + \det(\boldsymbol{A}_{\overline{a}_{r,c}}) \quad (2)$$

where

$(-1)^{r+c} \det(\boldsymbol{A}_{a_{r,c}})$	referred to as the *cofactor* of $\det(\boldsymbol{A})$ with respect to $a_{r,c}$;
$\det(\boldsymbol{A}_{\overline{a}_{r,c}})$	*remainder* of $\det(\boldsymbol{A})$ with respect to $a_{r,c}$;
determinant $\det(\boldsymbol{A}_{a_{r,c}})$	*minor* of $\det(\boldsymbol{A})$ with respect to $a_{r,c}$.

We note that the following two special cases of the expansion above are well known as *Laplace expansions* along row r and column c, respectively

$$\det(\boldsymbol{A}) = \sum_{r=1}^{n} a_{r,c}(-1)^{r+c} \det(\boldsymbol{A}_{a_{r,c}}) \quad (3)$$

$$\det(\boldsymbol{A}) = \sum_{c=1}^{n} a_{r,c}(-1)^{r+c} \det(\boldsymbol{A}_{a_{r,c}}). \quad (4)$$

C. Symbolic Analysis Problem of Analog Circuits

Consider a linear(ized) time-invariant analog circuit. Its system of equations can be formulated by, for example, the modified nodal analysis (MNA) approach in the following general form [44]:

$$\boldsymbol{T}\boldsymbol{x} = \boldsymbol{w}. \quad (5)$$

The *circuit unknown vector* $\boldsymbol{x} \in \mathcal{R}^n$ may be composed of node voltages and branch currents, and the *circuit matrix* $\boldsymbol{T}()$: $\mathcal{R}^n \rightarrow R^{n \times n}$ is a large **sparse** symbolic matrix, typically with just a few nonzero entries per row/column.

Symbolic analysis of analog circuits can be stated as the problem of solving the systems of symbolic equation (5), i.e., deriving the closed-form expression of a circuit unknown in terms of symbolic parameters in $\boldsymbol{T}$ and symbolic excitations expressed by $\boldsymbol{w}$. According to Cramer's rule, the kth component x_k of the unknown vector $\boldsymbol{x}$ is obtained as follows:

$$x_k = \frac{\sum_{i=1}^{n} w_i(-1)^{i+k} \det(\boldsymbol{T}_{t_{i,k}})}{\det(\boldsymbol{T})}. \quad (6)$$

Most symbolic simulators are targeted at finding various network functions, each function being defined as the ratio of an output unknown from $\boldsymbol{x}$ to an input from $\boldsymbol{w}$. These are special cases of (6) or the ratios of the two expressions in the form of (6).

Note that $(-1)^{i+k} \det(\boldsymbol{T}_{t_{i,k}})$ in (6) is the cofactor of $\det(\boldsymbol{T})$ with respect to element $t_{i,k}$ of matrix $\boldsymbol{T}$ at row i and column k. Therefore, the central issue in determinant-based symbolic analysis is how to find symbolic expressions of $\det(\boldsymbol{T})$ and the cofactors of $\det(\boldsymbol{T})$. In the rest of the paper, we focus on how to represent a symbolic determinant (Sections III and IV), and how to compute, manipulate, and evaluate a symbolic determinant (Section V). For simplicity, we assume in the paper that all the entries in $\boldsymbol{T}$ are distinct. This assumption has been used in previous symbolic analysis approaches; methods have been proposed to formulate the symbolic equations to meet (or closely meet) this assumption [21], [44].

III. ZBDD Representation of Symbolic Matrix Determinant

In this section, we apply the notation of ZBDD's to represent a symbolic matrix determinant. This leads to a new interpreted graph called DDD's. DDD's are a canonical representation for

matrix determinants, similar to BDD's for representing *binary functions* and ZBDD's for representing *subset systems*.

A key observation is that the circuit matrix is sparse, and many times, a symbolic expression may share many subexpressions. For example, consider the following determinant:

$$\det(\boldsymbol{M}) = \begin{vmatrix} a & b & 0 & 0 \\ c & d & e & 0 \\ 0 & f & g & h \\ 0 & 0 & i & j \end{vmatrix} = adgj - adhi - aefj - bcgj + cbih. \tag{7}$$

We note that subterms ad, gj, and hi appear in several product terms, and each product term involves a subset (four) out of ten symbolic parameters. Therefore, we view each symbolic product term as a subset, and use a ZBDD to represent the subset system composed of all the subsets each corresponding to a product term. Fig. 2 illustrates the corresponding ZBDD representing all the subsets involved in $\det(\boldsymbol{M})$ under ordering $a > c > b > d > f > e > g > i > h > j$. It can be seen that subterms ad, gj, and ih have been shared in the ZBDD representation.

Following directly from the properties of ZBDD's, we have the following observations. First, given a fixed order of symbolic parameters, all the subsets in a symbolic determinant can be represented uniquely by a ZBDD. Second, every rooted 1-path in the ZBDD corresponds to a product term, and the number of 1-edges in any rooted 1-path is n. The total number of rooted 1-paths is equal to the number of product terms in a symbolic determinant.

We can view the resulting ZBDD as a graphical representation of recursive application of determinant expansion formula (2) with the expansion order a, c, b, d, f, e, g, i, h, j. Each vertex is labeled with a matrix entry, and represents all the subsets contained in the corresponding submatrix determinant. The 1-edge points to the vertex representing all the subsets contained in the cofactor of the current expansion, and 0-edge points to the vertex representing all the subsets contained in the remainder.

To embed the signs of the product terms of a symbolic determinant into its corresponding ZBDD, we consider one step of matrix expansion with respect to $a_{r,c}$ as defined by (2). The sign is $(-1)^{r+c}$. Note that r and c are, respectively, the row and column indexes of the element $a_{r,c}$ in the submatrix before this step of expansion, say $\boldsymbol{A}'$. Let the *absolute* row and column indexes of the element $a_{r,c}$ in the original matrix $\boldsymbol{A}$ before any expansion be $r(a_{r,c})$ and $c(a_{r,c})$, respectively. Then, we observe that $(-1)^{r+c} = (-1)^{r+c-2}$ and $r+c-2$ is equal to the number of rows in $\boldsymbol{A}'$ with absolute indexes less than $r(a_{r,c})$ plus the number of columns in $\boldsymbol{A}'$ with absolute indexes less than $c(a_{r,c})$. We also note that all the rows and columns in $\boldsymbol{A}'$ except that of $a_{r,c}$ are represented in the subgraph rooted at the vertex pointed to by the 1-edge of vertex $a_{r,c}$. Therefore, the sign of a nonterminal vertex v, denoted by $s(v)$, can be defined recursively as follows.

1) Let $P(v)$ be the set of ZBDD vertices that originate the 1-edges in any 1-path rooted at v. Then

$$s(v) = \prod_{x \in P(v)} \operatorname{sign}\big(r(x) - r(v)\big)\operatorname{sign}\big(c(x) - c(v)\big) \tag{8}$$

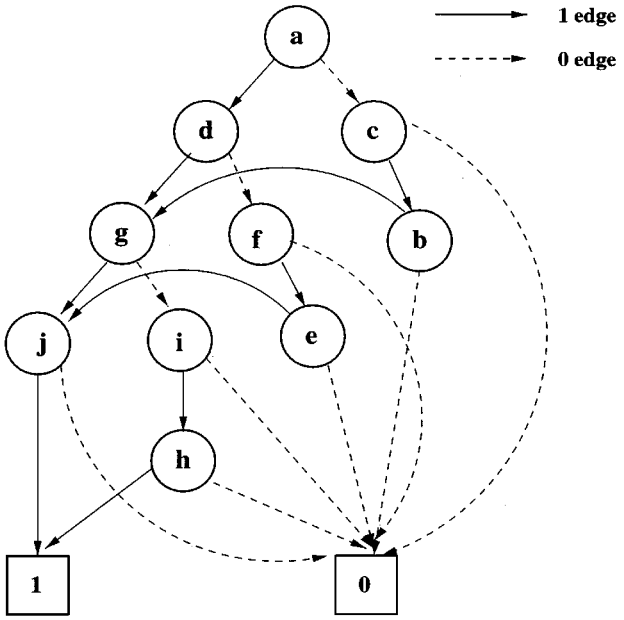


Fig. 2. A ZBDD representing $\{adgj, adhi, afej, cbgj, cbih\}$ under ordering $a > c > b > d > f > e > g > i > h > j$.

where $r(x)$ and $c(x)$ refer to the absolute row and column indexes of vertex x in the original matrix, and u is an integer so that

$$\operatorname{sign}(u) = \begin{cases} 1, & \text{if } u > 0, \\ -1, & \text{if } u < 0. \end{cases}$$

2) If v has an edge pointing to the 1-terminal vertex, then $s(v) = +1$.

This is called the *sign rule*. For example, in Fig. 3, shown aside by each vertex are the row and column indexes of that vertex in the original matrix, as well as the sign of that vertex obtained by using the sign rule above. We note that all the paths rooted at the same vertex yield the same vertex sign.

It can be verified that the product of all the signs in a rooted 1–path is exactly the sign of the corresponding product term. For example, consider the 1-path $acbgih$ in Fig. 3. The vertices that originate all the 1-edges are c, b, i, h, their corresponding signs are –, +, –, and +, respectively. Their product is +. This is the sign of the symbolic product term $cbih$.

With ZBDD's and the sign rule as two foundations, we are now ready to introduce formally our representation of a symbolic determinant. Let $\boldsymbol{A}$ be an $n \times n$ sparse matrix with a set of distinct m symbolic parameters $\{a_1, \cdots, a_m\}$, where $1 \le m \le n^2$. Each symbolic parameter a_i is associated with a unique pair $r(a_i)$ and $c(a_i)$, which denote, respectively, the row index and column index of a_i. A DDD is a signed, rooted, directed, acyclic graph with two terminal vertices, namely the 0-terminal vertex and the 1-terminal vertex. Each nonterminal vertex is labeled with a symbolic parameter a_i and the sign $s(a_i)$ determined by the sign rule defined by (8). It has two outgoing edges, called 1-edge and 0-edge. A vertex labeled by a_i represents a matrix determinant D defined recursively as follows.

1) If the vertex is the 1-terminal vertex, then $D = 1$.
2) If the vertex is the 0-terminal vertex, then $D = 0$.

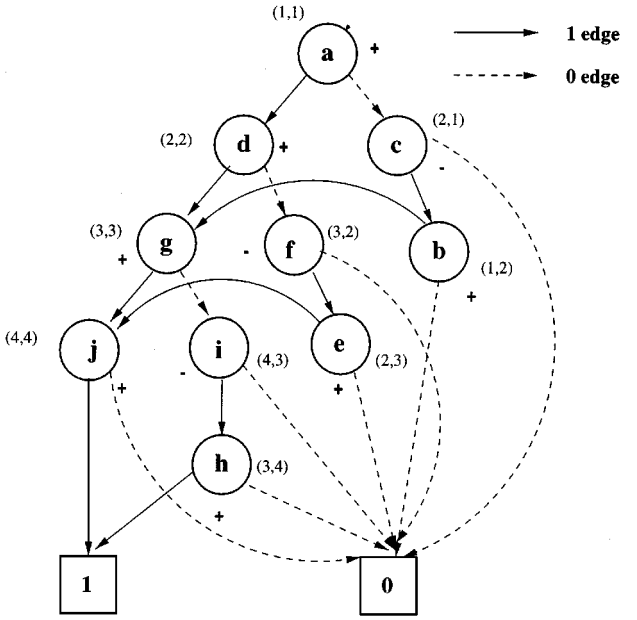


Fig. 3. A signed ZBDD for representing symbolic terms.

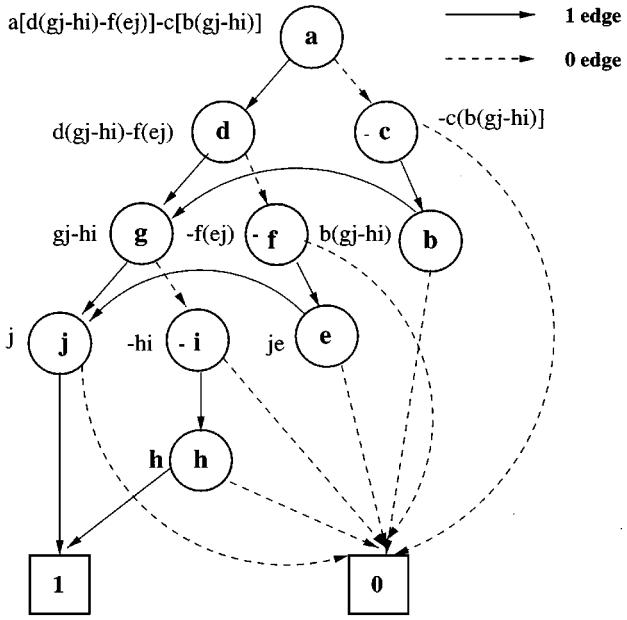


Fig. 4. A determinant decision diagram for matrix $\boldsymbol{M}$.

3) If the vertex is a nonterminal vertex, then $D = a_i s(a_i) D_{a_i} + D_{\overline{a}_i}$, where D_{a_i} ($D_{\overline{a}_i}$) is the matrix determinant represented by the vertex that was pointed to by the 1-edge (0-edge) of the vertex labeled by a_i.

Note that $s(a_i)D_{a_i}$ is the *cofactor* of D with respect to a_i, D_{a_i} is the *minor* of D with respect to a_i, $D_{\overline{a}_i}$ is the *remainder* of D with respect to a_i, and operations are algebraic multiplications and additions. For example, Fig. 4 shows the DDD representation of $\det(\boldsymbol{M})$ under ordering $a > c > b > d > f > e > g > i > h > j$.

To enforce the uniqueness and compactness of the DDD representation, the three rules of ZBDD's, namely, zero-suppression, ordered, and shared, described in Section II-A are adopted. This leads to DDD's having the following properties.

- Every 1-path from the root corresponds to a product term in the fully expanded symbolic expression. It contains exactly n 1-edges. The number of 1-paths from the root is equal to the number of product terms.
- For any determinant D, there is a unique DDD representation under a given vertex ordering.

We use |DDD| to denote the *size of* a DDD, i.e., the number of vertices in the DDD.

IV. An Effective Vertex-Ordering Heuristic

A key problem in many decision diagram applications is how to select a vertex ordering, since the size of the resulting decision diagram strongly depends on the chosen ordering. For example, if we choose vertex order $a > c > d > f > g > h > b > e > i > j$ for $\det(\boldsymbol{M})$ in Section III, then the resulting DDD is shown in Fig. 5. It has 13 vertices, in comparison to ten in Fig. 4, although they represent the same determinant. In this section, we describe an efficient heuristic for selecting a good vertex ordering, and show that it is optimal for a class of circuit matrices.

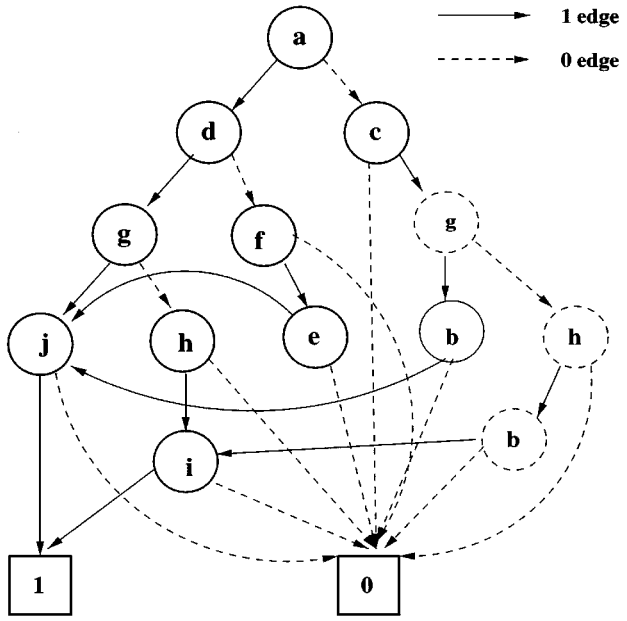


Fig. 5. DDD representing $\det(\boldsymbol{M})$ under ordering $a > c > d > f > g > h > b > e > i > j$.

We propose to select the vertex ordering for a DDD by examining the structure of the original matrix. Suppose that $\boldsymbol{A}$ is an $n \times n$ matrix with m nonzero elements (entries, or symbols). The vertex-ordering problem is how to *label* all the nonzero elements in $\boldsymbol{A}$ using integers one to m so that the resulting DDD constructed with the chosen order has a small size. As stated in Section II-A, those elements labeled by smaller integers will appear close to the leaves, or the bottom of the DDD, and the root is labeled by index m.

We propose a vertex-ordering heuristic, based on the refinement of a well-known strategy for Laplace expansion of a sparse matrix [21]. The basic idea is to label with larger indexes those columns or rows containing fewer nonzero

```
MATRIX_GREEDY_LABELING(A(p, q))
1   select column j (row i) that has least nonzero elements
2   s ← all the rows i (columns j) so that a_{i,j} ≠ 0
3   for all row i (column j) in s from the one with least nonzero elements
4       if there exist unlabeled elements in A(p − {i}, q − {j})
5           MATRIX_GREEDY_LABELING(A(p − {i}, q − {j}))
6   for all row i (column j) in s from the one with most nonzero elements
7       if a_{i,j} has not been labeled
8           label a_{i,j} by k and then increment k
```

Fig. 6. A DDD vertex-ordering heuristic.

elements. Elements in those dense rows and columns will be labeled using small indexes. Intuitively, this strategy increases the possibility of DDD subgraph sharing. Fig. 6 describes the proposed heuristic MATRIX_GREEDY_LABELING ($\boldsymbol{A}(p, q)$) for labeling all the elements in matrix $A(p, q)$. In the algorithm, $A(p-\{i\}, q-\{j\})$ denotes the matrix obtained from $A(p, q)$ by removing row i and column j. We keep a global counter k. Initially k is set to 1, and $p = q = e$.

As an example, consider how to apply MATRIX_GREEDY_LABELING to label the matrix $\boldsymbol{M}$ defined in Section III. The complete process is illustrated in Fig. 7. First, columns 1 and 4 of $\boldsymbol{M}$, as well as rows 1 and 4, have the least number of nonzero elements (2). We arbitrarily select column 1. Then the set of rows that have a nonzero element at column 1 are one and two, i.e., $s = \{1, 2\}$. Since row 1 has one nonzero element, and row 2 has two nonzero elements, lines 3–5 invoke first MATRIX_GREEDY_LABELING on matrix $\boldsymbol{M}$ after removing row 1 and column 1, then on matrix $\boldsymbol{M}$ after removing row 2 and column 1. The process is applied recursively on the resulting submatrices, and is illustrated in Fig. 7 from the top to the bottom. Then, elements are labeled in the Fig. 7 from the bottom to the top in the reverse order of expansion. These labels are marked in Fig. 7 at the top-right corner of each element. If we summarize all the labels assigned to the matrix elements using the original matrix structure, we have

$$\begin{pmatrix} a & b & 0 & 0 \\ c & d & e & 0 \\ 0 & f & g & h \\ 0 & 0 & i & j \end{pmatrix} \rightarrow \begin{pmatrix} 10 & 8 & 0 & 0 \\ 9 & 7 & 5 & 0 \\ 0 & 6 & 4 & 2 \\ 0 & 0 & 3 & 1 \end{pmatrix}.$$

The heuristic leads to compact DDD's for a large class of circuit matrices, as observed in our experiments described in Section VI. In the rest of this section, we show that the heuristic is optimal for a class of circuit matrices, called *tridiagonal matrices*. Tridiagonal matrices are matrices that have only nonzero elements at positions (i, i), $(i-1, i)$, and $(i+1, i)$. They have the structure as shown in (8a), found at the bottom of the page.

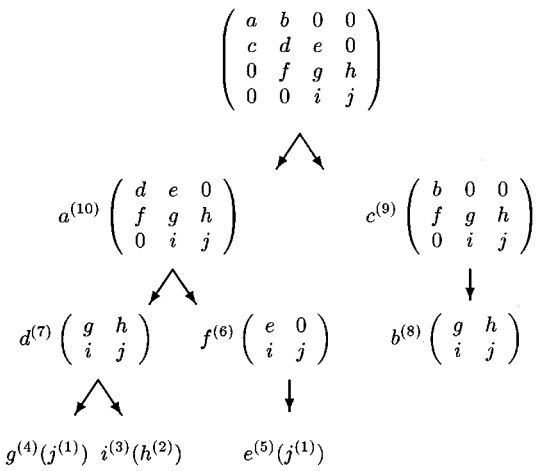


Fig. 7. An illustration of DDD vertex-ordering heuristic.

The number of nonzero elements in $\boldsymbol{A}_{n+1,n+1}$ is $3n+1$. Matrix $\boldsymbol{M}$ used throughout this paper is a 4×4 tridiagonal matrix.

To show that the algorithm MATRIX_GREEDY_LABELING yields an optimum ordering, we note that the lower bound on the number of DDD vertices is equal to the number of matrix entries, i.e., each vertex appears only once in the final DDD. We will show that, for tridiagonal matrices, the ordering given by MATRIX_GREEDY_LABELING results in a DDD with the number of vertices equal to the number of nonzero matrix elements. This is proved by induction. It is easy to see that the result is true for 1×1 and 2×2 matrices. Now we assume that it is true for $\boldsymbol{A}_{n,n}$, i.e., the number of DDD vertices is $3n-2$. We prove that it is also true for $\boldsymbol{A}_{n+1,n+1}$: the number of DDD vertices is $3(n+1)-2$. Let vertex $D_{n-1,n-1}$ represent the DDD of $\det(\boldsymbol{A}_{n-1,n-1})$ and $D_{n,n}$ represent the DDD of $\det(\boldsymbol{A}_{n,n})$. Matrix $\boldsymbol{A}_{n+1,n+1}$ has an extra row and column with three nonzero elements $a_{n+1,n+1}$, $a_{n,n+1}$, and $a_{n+1,n}$. Algorithm MATRIX_GREEDY_LABELING assigns integer labels $3n+1$, $3n$ and $3n-1$ to elements $a_{n+1,n+1}$, $a_{n,n+1}$, and $a_{n+1,n}$, respectively. This gives ordering $a_{n+1,n+1} > a_{n,n+1} > a_{n+1,n} > a_{n,n}$. We, thus, first create a DDD vertex labeled by $a_{n+1,n+1}$. Its 1-edge points to vertex $D_{n,n}$, and its 0-edge points to the vertex that corresponds to $a_{n,n+1}$. The 0-edge of vertex $a_{n,n+1}$ points to the 0-terminal. Its 1-edge points to the vertex that represents the determinant of matrix $\boldsymbol{A}_{n+1,n+1}$ after removing the first column and the second row. Since the first row in the resulting matrix contains only one nonzero element $a_{n+1,n}$, we can create a DDD vertex for $a_{n+1,n}$ with its 1-edge pointing to

$$\boldsymbol{A}_{n+1,n+1} = \begin{pmatrix} a_{n+1,n+1} & a_{n+1,n} & 0 & \cdots & \cdots & \cdots & 0 \\ a_{n,n+1} & a_{n,n} & a_{n,n-1} & 0 & \cdots & \cdots & 0 \\ 0 & a_{n-1,n} & a_{n-1,n-1} & a_{n-1,n-2} & 0 & \cdots & 0 \\ 0 & 0 & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & \cdots & 0 & a_{3,4} & a_{3,3} & a_{3,2} & 0 \\ 0 & \cdots & \cdots & 0 & a_{2,3} & a_{2,2} & a_{2,1} \\ 0 & \cdots & \cdots & \cdots & 0 & a_{1,2} & a_{1,1} \end{pmatrix} \tag{8a}$$

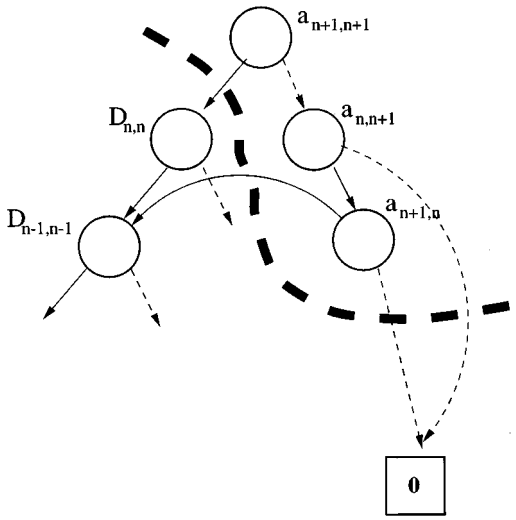


Fig. 8. An illustration of DDD construction for tridiagonal matrices.

$D_{n-1,\,n-1}$, and its 0-edge pointing to the 0-terminal. This is illustrated in Fig. 8. Only three vertices are added for representing $\boldsymbol{A}_{n+1,\,n+1}$. The total number of DDD vertices for $\boldsymbol{A}_{n+1,\,n+1}$ is, thus, $3n-2+3=3(n+1)-2$. Our conjecture is, therefore, proved.

We have proved that for tridiagonal matrix $\boldsymbol{A}_{n,\,n}$, the number of DDD vertices is $3n-2$. As a comparison, the number of expanded product terms in $\det(\boldsymbol{A}_{n,\,n})$ is $F(n+1)$, where $F(i)$ is the ith Fibonacci number defined by

$$F(n)=F(n-1)+F(n-2)\quad n>2,$$
$$F(2)=F(1)=1.$$

Each product term involves n symbols. To store all the product terms without considering sharing, the memory requirement is proportional to $nF(n+1)$. Further, any symbolic manipulation using the expended form would have time complexity at best proportional to $nF(n+1)$. In Fig. 9, the number of DDD vertices is plotted against the number of product terms.

The practical relevance of tridiagonal matrices is that they correspond to the circuit matrices for ladder networks—an important class of circuit structures in analog design. A three-section ladder circuit is shown in Fig. 10. The system of equations can be formulated as shown in (9) at the bottom of the page.

If we view each entry as a distinct symbolic parameter, the resulting circuit matrix is a tridiagonal matrix. We note that many practical circuits have the structure of ladders or close to ladders.

We emphasize that just like the BDD representation for Boolean functions, in the worst case, the number of DDD vertices can grow exponentially with the size of a circuit. Nevertheless, as we have observed (in Section VI) that with the proposed vertex-ordering heuristic, the numbers of vertices in the resulting DDD's are reasonable for practical analog circuits.

V. Manipulation and Construction of Determinant Decision Diagrams

In this section, we show that, using determinant decision diagrams, algorithms needed for symbolic analysis and its applications can be performed with the time complexity proportional to the size of the diagrams being manipulated, **not the number of rooted 1-paths in the diagrams, i.e., product terms in the symbolic expressions**. Hence, as long as the determinants of interest can be represented by reasonably small graphs, our algorithms are quite efficient.

A basic set of operations on matrix determinants is summarized in Table I. Most operations are simple extensions of subset operations introduced by Minato on ZBDD's [29]. These few basic operations can be used directly and/or combined to perform a wide variety of operations needed for symbolic analysis. In this section, we first describe these operations, and then use an example to illustrate the main ideas of these operations and how they can be applied to compute network function sensitivities—a key operation needed in optimization and testability analysis. We also show that the generation of significant product terms can be casted as the k-shortest path problem in a DDD and solved elegantly in time $O(k\cdot|\mathrm{DDD}|)$.

A. Implementation of Basic Operations

We summarize the implementation of these operations in Fig. 11. For the clarity of the description, the steps for computing the signs associated with DDD vertices, using the sign rule defined in Section III, are not shown.

As the basis of implementation, we employ two techniques originally developed by Brace, Rudell and Bryant for implementing decision diagrams efficiently [7]. First, a basic procedure GetVertex(top, D_1, D_0) is to generate (or copy) a vertex for a symbol top and two subgraphs D_1 and D_0. In the procedure, a hash table is used to keep each vertex unique; vertex elimination and sharing are managed mainly by GetVertex. With GetVertex, all the major operations for DDD's in Table I are described in Fig. 11.

$$\begin{pmatrix} \frac{1}{R_1} & -\frac{1}{R_1} & 0 & 0 \\ -\frac{1}{R_1} & \frac{1}{R_1}+\frac{1}{R_2}+\frac{1}{R_3} & -\frac{1}{R_3} & 0 \\ 0 & -\frac{1}{R_3} & \frac{1}{R_3}+\frac{1}{R_4}+\frac{1}{R_5} & -\frac{1}{R_5} \\ 0 & 0 & -\frac{1}{R_5} & \frac{1}{R_5}+\frac{1}{R_6} \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{pmatrix} = \begin{pmatrix} I_{in} \\ 0 \\ 0 \\ 0 \end{pmatrix} \tag{9}$$

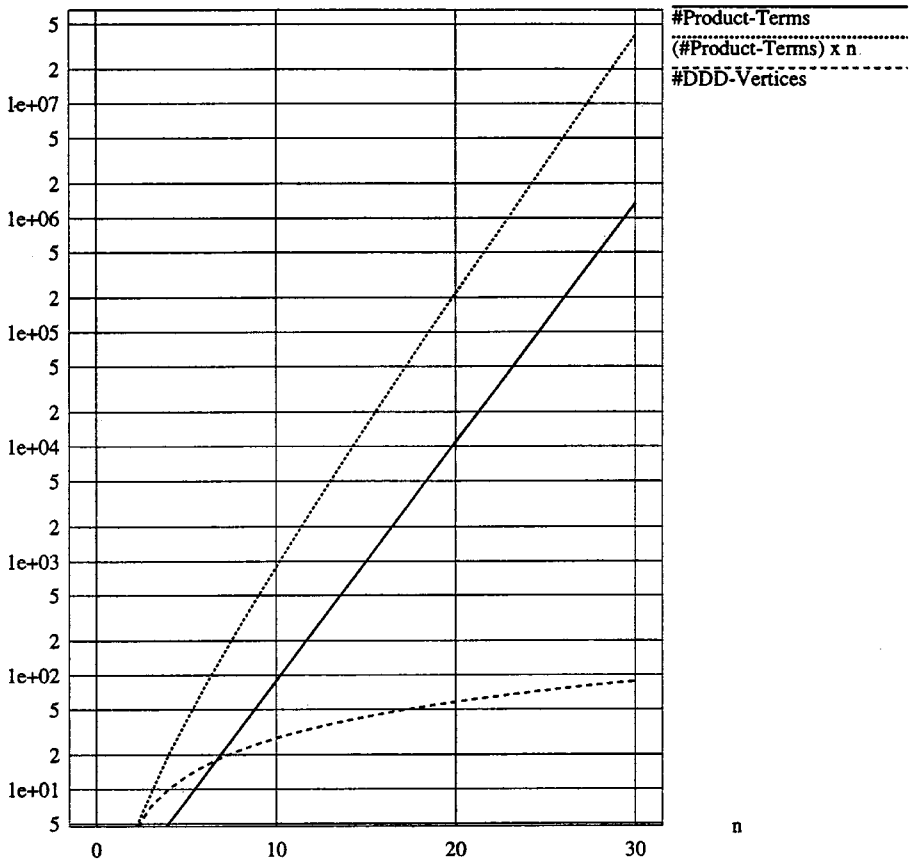


Fig. 9. A comparison of DDD sizes versus numbers of product terms for tridiagonal matrices.

TABLE I
SUMMARY OF BASIC OPERATIONS

Determinant operation	Result	Subset operation
VERTEXONE()	return 1	Base()
VERTEXZERO()	return 0	Empty()
COFACTOR(D, s)	return the cofactor of D wrt s	Subset1(D, s)
REMAINDER(D, s)	return the remainder of D wrt s	Subset0(D, s)
MULTIPLY(D, s)	return $s \times D$	Change(D, s)
TERMSUBTRACT(D, P)	return $D - P$ where P is a product term in D	Diff(D, P)
DDD_OF_MATRIX$(\mathbf{A})$	construct the DDD for matrix $\mathbf{A}$	
EVALUATE(D)	return the numerical value of D	

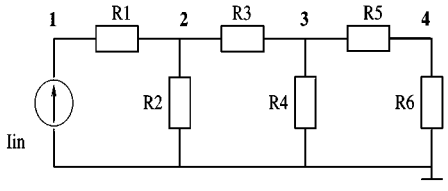


Fig. 10. A three-section ladder circuit.

Second, similar to conventional BDD's, we use a cache to remember the results of recent operations, and refer to the cache for every recursive call. In this way, we can avoid duplicate executions for equivalent subgraphs. This enables us to execute these operations in a time that is (almost) linearly proportional to the size of a graph.

Evaluation: Given a determinant decision diagram rooted at a vertex D and a set of numerical values for all the symbolic parameters, EVALUATE(D) computes the numerical value of the corresponding matrix determinant. EVALUATE(D) naturally exploits subexpression sharing in a symbolic expression, and has time complexity (almost) *linear* in the size of the diagram.

Construction: Let $\boldsymbol{A}(e, e)$ be an n-by-n symbolic matrix, where e is a set of integers from one to n. DDD_OF_MATRIX $(\boldsymbol{A}(p, q))$ constructs a determinant decision diagram of a submatrix of $\boldsymbol{A}$ with row set $p \subseteq e$ and column set $q \subseteq e$ such that $|p| = |q|$ for a given ordering of symbolic parameters. It can be viewed as a generalized Laplace expansion procedure of matrix determinants. In line 3 of the procedure DDD_OF_MATRIX, a nonzero element s is selected, and the determinant is expanded. Due to the canonicity of DDD's, s can be any nonzero matrix element, and the resulting DDD is always the same. However, the best expansion order is to use the element with the largest integer label in line 3 of the procedure DDD_OF_MATRIX.

```
COFACTOR(D, s)
1  if (D.top < s) return VERTEXZERO()
2  if (D.top = s) return D_1
3  if (D.top > s) return GETVERTEX(D.top, COFACTOR(D_0, s), COFACTOR(D_1, s))

REMAINDER(D, s)
1  if (D.top < s) return D
2  if (D.top = s) return D_0
3  if (D.top > s) return GETVERTEX(D.top, REMAINDER(D_0, s), REMAINDER(D_1, s))

MULTIPLY(D, s)
1  if (D.top < s) return GETVERTEX(s, 0, D)
2  if (D.top = s) return GETVERTEX(s, D_1, D_0)
3  if (D.top > s) return GETVERTEX(D.top, MULTIPLY(D_0, s), MULTIPLY(D_1, s))

TERMSUBTRACT(D, P)
1  if (D = 0) return VERTEXZERO()
2  if (P = 0) return D
3  if (D = P) return VERTEXZERO()
4  if (D.top > P.top) return GETVERTEX(D.top,TERMSUBTRACT(D_0, P), D_1)
5  if (D.top < P.top) return TERMSUBTRACT(D, P_0)
6  if (D.top = P.top)
       return GETVERTEX(D.top, TERMSUBTRACT(D_0, P_0), TERMSUBTRACT(D_1, P_1))

DDD_OF_MATRIX(A(p, q))
1  if (A = 0) return VertexZero()
2  if (A = 1) return VertexOne()
3  let s be a nonzero element at row i and column j of A
4      return GETVERTEX(s, DDD_OF_MATRIX(A(p - {i}, q - {j})), DDD_OF_MATRIX(A|_{s=0}))

EVALUATE(D)
1  if (D = 0) return 0
2  if (D = 1) return 1
3  return EVALUATE(D_0) + s(D) * D.top * EVALUATE(D_1)
```

Fig. 11. Implementation of basic operations for symbolic analysis and applications.

Cofactor and Derivative: COFACTOR(D, s) is to compute the cofactor of a symbolic determinant represented by a DDD vertex D with respect to symbolic parameter s. COFACTOR is perhaps the most important operation in symbolic analysis of analog circuits. For example, the network functions can be obtained by first computing some cofactors, and then combining these cofactors according to some rules (Cramer's rule).

B. Illustration of Basic Operations and its Use in Circuit Sensitivity

In this subsection, we use an example to show how the network function sensitivity can be computed using DDD-based COFACTOR. We also use COFACTOR to exemplify the main ideas of a typical DDD-based operation.

Consider the ladder circuit shown in Fig. 10. Its system of equations has been formulated in (9). The input impedance Z_{in} is defined as

$$Z_{\text{in}} = \frac{v_1}{I_{\text{in}}}.$$

If each matrix entry is viewed as a distinct symbol, the determinant of the circuit matrix can be rewritten as (7). We redraw its DDD in Fig. 12(a), where the 0-terminal and all the 0-edges pointing to the 0-terminal are suppressed. In Fig. 12(a), for each vertex labeled by a lower-case letter, we use the corresponding upper-case letter to denote the determinant represented by that vertex. The root of the DDD represents the determinant, denoted by A, of the circuit matrix. Note that $a = 1/R_1$ and $d = (1/R_1) + (1/R_2) + (1/R_3)$. From Cramer's rule

$$v_1 = \frac{I_{\text{in}} \times \text{COFACTOR}(A,\, a)}{A}.$$

Thus

$$Z_{\text{in}} = \frac{\text{COFACTOR}(A,\, a)}{A}.$$

We consider the normalized sensitivity of the input impedance Z_{in} with respect to resistor R_2

$$\begin{aligned} S_{R_2}^{Z_{\text{in}}} &= \left(\frac{R_2}{Z_{in}}\right)\left(\frac{\partial Z_{in}}{\partial R_2}\right) \\ &= \left(\frac{R_2 A}{\text{COFACTOR}(A,\, a)}\right)\frac{\partial}{\partial d}\left(\frac{\text{COFACTOR}(A,\, a)}{A}\right) \\ &\quad \cdot \left(\frac{\partial d}{\partial R_2}\right) \\ &= \left(\frac{R_2 A}{\text{COFACTOR}(A,\, a)}\right) \\ &\quad \cdot \left(-\frac{\text{COFACTOR}(A,\, a)}{A^2}\frac{\partial A}{\partial d} \right. \\ &\qquad \left. + \frac{1}{A}\frac{\partial \text{COFACTOR}(A,\, a)}{\partial d}\right)\left(-\frac{1}{R_2^2}\right). \end{aligned}$$

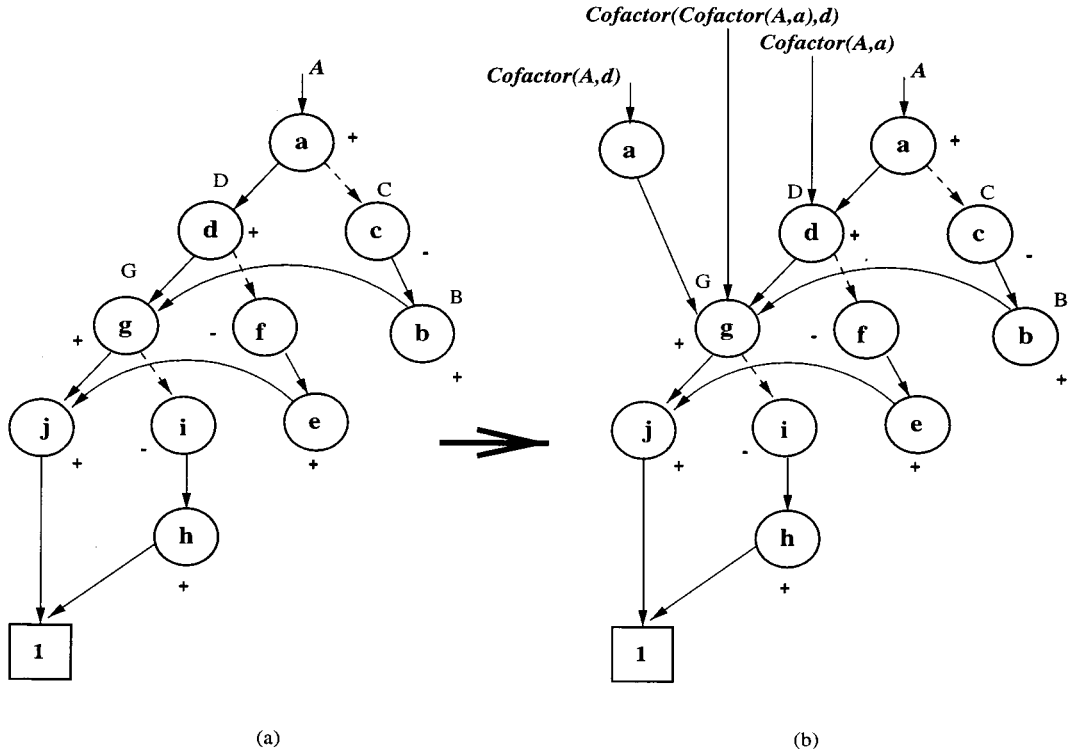


Fig. 12. DDD-based derivation of cofactors.

Note that

$$\frac{\partial A}{\partial d} = \textsc{Cofactor}(A, d)$$

and

$$\frac{\partial \textsc{Cofactor}(A, a)}{\partial d} = \textsc{Cofactor}(\textsc{Cofactor}(A, a), d)$$

we have

$$S_{R_2}^{Z_{\text{in}}} = \left(\frac{1}{R_2}\right)\left(\frac{\textsc{Cofactor}(A, d)}{A} - \frac{\textsc{Cofactor}(\textsc{Cofactor}(A, a), d)}{\textsc{Cofactor}(A, a)}\right).$$

The three cofactors $\textsc{Cofactor}(A, a)$, $\textsc{Cofactor}(A, d)$, and $\textsc{Cofactor}(\textsc{Cofactor}(A, a), d)$ in the expression above can be computed elegantly using algorithm COFACTOR in Fig. 11 on the DDD shown in Fig. 12(a). Recall that the vertex ordering in Fig. 12(a) is $a > c > b > d > f > e > g > i > h > j$.

First, consider how to compute $\textsc{Cofactor}(A, a)$. Since $A.\text{top} = a = s$, $A_1 = D$ is returned. $\textsc{Cofactor}(A, a)$ points to D. Similarly, $\textsc{Cofactor}(\textsc{Cofactor}(A, a), d)$ points to G. They are shown in Fig. 12(b).

Next consider $\textsc{Cofactor}(A, d)$. Since $A.\text{top} = a > d$, line 3 of the algorithm is executed with $A_0 = C$ and $A_1 = D$; i.e., $\textsc{GetVertex}(a, \textsc{Cofactor}(C, d), \textsc{Cofactor}(D, d))$. Then the procedure is invoked recursively, respectively, for $\textsc{Cofactor}(C, d)$ and $\textsc{Cofactor}(D, d)$. This process is shown in Fig. 13, where top-down solid arrows illustrate the recursive invocation of the procedure COFACTOR, bottom-up dashed arrows show how the final result is synthesized, and each step is labeled by a number that indicates its order of execution. Note that $\textsc{GetVertex}(s, 0, 0)$ returns the 0-terminal based on the zero-suppression rule. Eventually, $\textsc{Cofactor}(A, d)$ returns $\textsc{GetVertex}(a, 0, G)$. Since no vertex exists with label a, the 0-edge pointing to zero and the 1-edge pointing to d, $\textsc{GetVertex}(a, 0, G)$ will create a new vertex as shown in Fig. 12(b). During the recursive process, $\textsc{Cofactor}(0,d)$ is first calculated at Steps 3 and 4. Later on at Step 6, its return value has been cached and is used directly. This avoids the duplicate execution of COFACTOR on the same subgraph.

All three cofactors and the original determinant are compactly represented in a **single** four-root DDD as shown in Fig. 12(b). From this DDD, the sum-of-product expressions of cofactors and determinants can be generated efficiently by enumerating all its corresponding rooted 1-paths. For this example, we have $\textsc{Cofactor}(A, d) = agj - aih$, $\textsc{Cofactor}(\textsc{Cofactor}(A, a), d) = G = gj - ih$, and $\textsc{Cofactor}(A, a) = D = dgj - dih - fej$. The DDD representation enables efficient computation of exact sum-of-product symbolic expressions and their sensitivities. We can also generate the sequence-of-expression representations by introducing one intermediate symbolic symbol for each vertex. In comparison, sensitivity computation using directly the sequence-of-expressions approach requires grammar-driven compilation [27].

C. Generation of Significant Terms

Many small-signal characteristics are dominated by a small number of product terms. This has been observed and exploited previously in the context of transfer function approximation. Many times, analog designers are interested in the symbolic expressions of the first few dominating terms. In our framework, the extraction of significant product terms can be transformed to

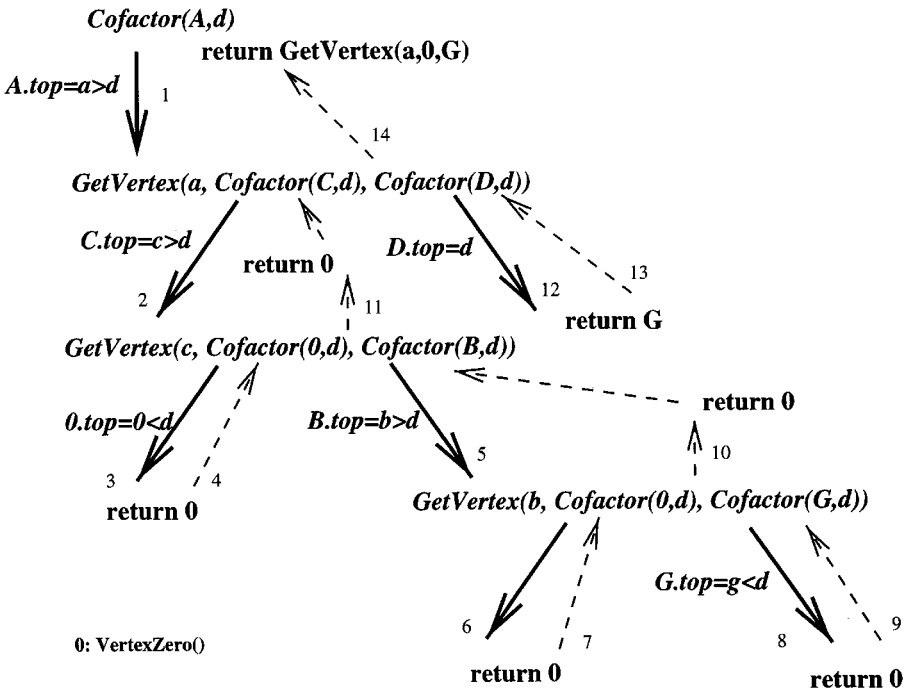


Fig. 13. Illustration of DDD-based cofactoring.

the problem of finding k shortest paths in a DDD, and solved elegantly by an $O(k|\mathrm{DDD}|)$ algorithm. We note that the problem itself can be solved efficiently by matroid-intersection-based methods [45], [46], [51], [52].

We adapted the weighting scheme of Yu and Sechen [51]. A 1-edge originated from vertex a_i is assigned weight $-\log|a_i|$, where $|a_i|$ is the numerical value of symbolic parameter a_i. All the 0-edges are assigned weight 0. Then the *cost* of a path in a DDD is defined to be the total weights of all the edges along the path. With this, the most significant product term in a symbolic determinant D corresponds to the minimum cost (shortest) path between the DDD root and the 1-terminal. The shortest path in a DDD can be obtained by a depth-first search, which has the time complexity $O(|\mathrm{DDD}|)$ [15].

A nice property of DDD's is that after we find the shortest path from a DDD, we can subtract it from the DDD using DDD operation TERMSUBTRACT. We can find the next shortest path in the resulting DDD. In this manner, we can find the k shortest paths in time $O(k \cdot |\mathrm{DDD}|)$.

VI. Experimental Results

We have implemented in C++ the proposed symbolic analysis algorithms, and tested our program on a set of circuits varying from RLC filters to bipolar and MOS integrated circuits. The set of test circuits includes

- *millerOpamp*, a two-stage miller compensated MOS opamp from [21];
- μA741, a bipolar opamp containing 26 transistors and 11 resistors, with the schematic in Fig. 15;
- *cascodeOpamp*, a CMOS cascode opamp containing 22 transistors with the schematic in Fig. 16,
- *ladder7, ladder21, ladder100,* 7-, 21-, and 100-section cascade resistive ladder networks;
- *rctree1, rctree2*, two RC tree networks;
- some RLC filters, named *butter, rlctest, vcstst, ccstest*, and *bigtst*.

For nonlinear integrated circuits, DC analysis is first performed using *SPICE*, and the resulting small-signal models from the output of *SPICE* are used in symbolic analysis. For the completeness, the small-signal models used for bipolar and MOS transistors are described in Fig. 14(a) and (b), respectively. The MNA approach as used in *SPICE* is employed to formulate the circuit equations. Exact symbolic expressions for (voltage) transfer functions ($V_{\mathrm{out}}/V_{\mathrm{in}}$) are computed using Cramer's rule and shared DDD representations of symbolic determinants and cofactors. The transfer function is in the form of the ratio of two DDD's, which are represented compactly using a **single shared** DDD with two roots; this is referred to as the DDD representation of the transfer function.

Table II[3] describes the statistics of all the test circuits, the resulting DDD sizes, and *SCAPP* results. Columns 2–6 describe, respectively, the number of nodes in each circuit, the number of nonzero elements in each circuit matrix, the number of product terms in the transfer function and the numbers of vertices in the DDD representation of the transfer function without and with the use of the vertex-ordering heuristic described in Fig. 6. For each circuit, the number of DDD vertices without vertex-ordering is the average of that of ten randomly generated orderings. *SCAPP* is used to generate the sequence of expressions (in the C program) for each transfer function. The number of multiplications (divisions are counted as multiplications), the number of additions, and the number of intermediate expressions used for computing each transfer function are reported in Columns 7–9. If the sequence of expressions is generated from the DDD

[3]In [38], only the statistics of the determinants of the circuit matrices are reported. Here, the statistics are collected for the transfer functions in order to compare with *SCAPP* which calculates the transfer functions.

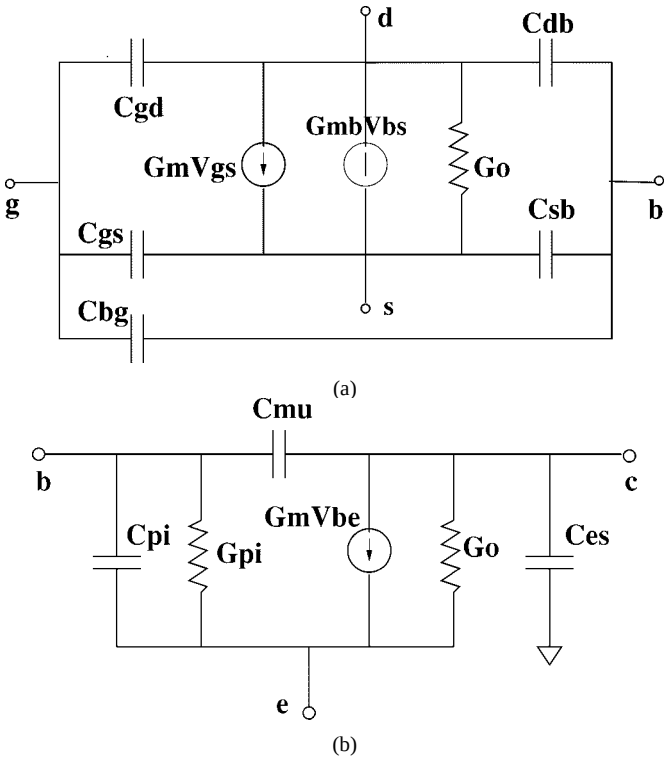


Fig. 14. (a) MOSFET small-signal model and (b) bipolar transistor model.

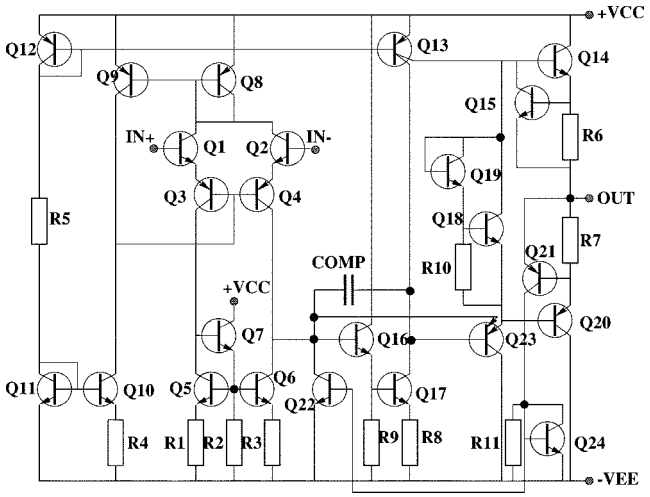


Fig. 15. The circuit schematic of bipolar μA741.

representation of the transfer function, each vertex will introduce one multiplication and one addition (if its 0-edge does not point to the 0-terminal) and, hence, the total number of multiplications and the total number of additions are bounded by the number of DDD vertices.

From Table II, we can make several observations.

1) Ordering leads to DDD's significantly smaller than that of nonordering. For ladder networks *ladder7*, *ladder21*, and *ladder100*, the numbers of DDD vertices are exactly the numbers of nonzero matrix elements.
2) For a small circuit, the number of DDD vertices may be greater than the number of product terms. This is not surprising, since the number of DDD vertices without exploiting sharing would be the number of product terms times the number of symbolic parameters in each product term.
3) For large circuits, the numbers of DDD vertices can be several orders of magnitude smaller than the numbers of product terms. Further, the difference becomes more dramatic with the increase of the circuit size.
4) The sequences of expressions generated from DDD's use about two thirds of multiplications required by *SCAPP* for ladder-structured circuits (*ladder7*, *ladder21*, and *ladder100*), and use less than half the multiplications of *SCAPP* for tree-structured circuits (*rctree1* and *rctree2*). It may use much more multiplications than *SCAPP* for those circuits that do not have ladder- or tree-like structures ($cascodeOpamp$ and $\mu A741$).[4]

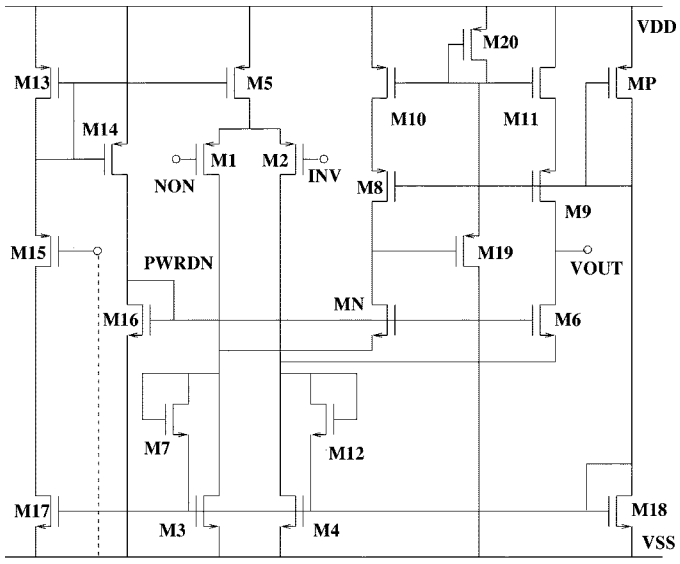


Fig. 16. The circuit schematic of MOS cascode Opamp.

We then compare our program with *ISAAC* [21] and *Maple-V* [10] for generating the complete sum-of-product expressions of the transfer functions. *ISAAC* is a well-known special-purpose symbolic analyzer designed for analog integrated circuits. *Maple-V* is a general-purpose mathematic package capable of solving linear equations symbolically. To use *Maple-V*, we use our program to set up the circuit equations and then feed the circuit matrices to *Maple-V*. The results are described in Table III. All data are obtained using a SUNsparc 20 with 32M memory. We can observe that our program runs significantly faster than both *ISAAC* and *Maple-V*, and uses much less memory. For slightly large circuits, both *ISAAC* and *Maple-V* ran out of memory. The symbolic expressions generated by *ISAAC*, *Maple-V* and the DDD-based approach are the same for each circuit.

We have implemented a frequency-domain circuit simulator by evaluating the computed DDD-represented transfer functions and compared the results with *SPICE*. For all the test circuits, our simulator produced the same numerical outputs as those of *SPICE*. Note that *SPICE* employs pivoting to improve the nu-

[4]Very recently, we showed that by exploiting the design hierarchy and automated partitioning, the DDD-based approach can generate even more compact representations than that of *SCAPP* [41], [42].

TABLE II
COMPARISON OF DDD SIZES WITH/WITHOUT VERTEX-ORDERING AND *SCAPP*

ckt name	#nodes	#symbols (matrix)	#terms	$\|DDD\|$ w/o ordering	$\|DDD\|$ w/ ordering	*SCAPP* #mul	*SCAPP* #add	*SCAPP* #expr
millerOpamp	7	21	29	95.2	51	36	60	44
butter	8	19	14	57.1	22	30	21	39
ladder7	9	22	22	85.6	26	36	25	46
rlctest	10	37	200	634.2	152	208	161	219
ccstest	10	35	176	510.8	97	208	162	219
vcstst	11	46	120	435.8	70	238	169	252
ladder21	23	64	17712	53434.4	84	116	79	140
cascodeOpamp	15	77	119395	150868.1	1994	444	596	460
$\mu A741$	24	89	119011	–*	6654	198	365	233
bigtst	33	112	1.6×10^7	–	951	996	715	1030
ladder100	102	301	5.7×10^{20}	–	398	594	398	697
rctree1	41	119	7.1×10^7	–	208	508	311	550
rctree2	54	158	3.0×10^{10}	–	299	660	407	715

-*: out of memory.

TABLE III
COMPARISON OF THE PROPOSED ALGORITHM AGAINST *ISAAC* AND *Maple-V*

circuit	*ISAAC* CPU time (seconds)	*ISAAC* Memory (bytes)	*Maple-V* CPU time (seconds)	*Maple-V* Memory (bytes)	Proposed Algorithm CPU time (seconds)	Proposed Algorithm Memory (bytes)
butter	0.61	493k	0.03	10.9k	0.033	8.1K
ladder7	1.05	980k	0.63	250k	0.033	8.1k
millerOpamp	0.49	640k	0.6	250k	0.066	24.5k
rlctest	–*		17.3	15.2M	0.10	49.1k
vcstst	–	–	104.3	23.1M	0.15	112.1k
ladder21	–	–	–	–	0.066	32.7k
cascodeOpamp	–	–	–	–	7.62	4.9M
$\mu A741$	–	–	–	–	9.13	5.2M
bigtst	–	–	–	–	2.81	2.6M

–*: out of memory.

merical accuracy of the solution of a system of linear equations, whereas no special consideration has been given to the numerical aspect in our DDD-based approach. We further observe that, in comparison with *SPICE* which uses numerical **LU** decomposition, and numerical evaluation with the sequences of expressions generated by *SCAPP*,[5] our DDD-based simulator has the following interesting features.

- Evaluation of DDD determinants and cofactors uses only multiplications, additions, subtractions, and **no** divisions. The notorious "divided-by-zero" problem does not occur in DDD evaluation.
- All multiplications in DDD evaluation are performed between a derived value (the value of a cofactor) and one from the original problem (matrix entry). Further, the *depth* of derived operation, i.e., the number of (nested) multiplications required to obtain the final value from a value in the original problem, is at most n. In contrast, in **LU** decomposition, most times, operations are performed among two derived values, and the depth of derived operation is at least $2n$.
- DDD's are constructed in such a way to achieve maximal sharing of subexpressions. As a consequence, those values close in their magnitudes are likely to be manipulated together.

Table IV shows the comparison of our DDD-based simulator with *SPICE* in terms of CPU time for repetitive numerical evaluation. For each circuit, 1000 frequency points were simulated.

TABLE IV
COMPARISON OF FREQUENCY ANALYSIS BY THE PROPOSED ALGORITHM AND BY *SPICE*

circuit	*SPICE* (seconds)	DDD-based numerical evaluation (seconds)
butter	0.96	0.26
millerOpamp	0.36	0.28
rcltest	0.71	0.65
ladder21	0.96	0.75
bigtst	3.68	2.78
cascodeOpamp	3.16	11.8

At each frequency point, the DDD-represented transfer function was evaluated by first computing the numerical value of each matrix entry from the values of circuit parameters and frequency, and then substituting the computed value of each entry to compute the DDD values. The proposed algorithm is actually faster than *SPICE* for small circuits, but is slower for large circuits. We note that the complexity of such *repetitive numerical evaluation* is linearly proportional to the number of DDD vertices, but the number of DDD vertices may grow exponentially with the size of a circuit. Sparse-matrix-based numerical **LU** decomposition has been observed to run in $O(n^{1.1-1.5})$ for typical circuits [33]. Therefore, the straightforward use of exact symbolic expressions—DDD-based or even *SCAPP*'s sequences of expressions—for numerical evaluation may not offer

[5]We note that hierarchical symbolic analysis employed in *SCAPP* is essentially partial symbolic LU decomposition by Gaussian elimination [44].

any speed advantage over fine-tuned numerical simulators such as *SPICE*. However, DDD-based symbolic analysis may still be attractive for repetitive numerical evaluation, since it allows the other "latency" properties of symbolic expressions to be exploited efficiently. For example, for the frequency-domain simulation of time-invariant circuits, all the circuit parameters but the complex variable s remain unchanged for all the frequency points, and, therefore, the use of s-expanded symbolic expressions can provide a significant speedup over *SPICE*. We have shown that s-expanded symbolic expressions can be derived very efficiently using DDD's and then only one DDD evaluation is needed for all the frequency points; this speeds up *SPICE* significantly [39].

VII. Related Work

The proposed approach is an application of decision diagram concepts to symbolic network analysis. In this section, we summarize some closely related work in these two areas. We refer the reader to [21] and [26] for comprehensive surveys of symbolic analysis techniques and applications, and [30] and [36] for decision diagrams.

A. Comparison with Existing Determinant-Based Symbolic Techniques

Previously, determinant expansion has been exploited for symbolic analysis of analog circuits. The work includes the parameter extraction method [2], the algebraic formulation method [35], and recursive Laplace expansion with minor storage and row/column ordering as implemented in *ISAAC* [21]. Parameter extraction was developed for handling large sparse matrices with a few symbolic entries and many numerical entries. The key idea is to apply a refined form of determinant expansion in (2) on all the symbolic entries first, then to use any standard numerical method to evaluate the values of minors that contain only numerical entries. This idea can be naturally incorporated into DDD's where symbolic entries are labeled first and numerical entries are labeled after symbolic entries (with small indexes). With this, numerical entries will appear at the bottom of a DDD, which can be evaluated and condensed. The incorporation of parameter extraction into DDD's will lead to a new method capable of handling large sparse matrices with both numerical and (potentially many) symbolic entries. In contrast, it is generally difficult to combine parameter extraction with other symbolic methods.

The algebraic formulation method of Sannuti and Puri exploits the structure of determinants and circuits to establish the condition for valid nonzero product terms as expressed in (1). Then the valid product terms are enumerated.

The DDD-based approach inherits many ideas found in recursive Laplace expansion with minor storage and row/column ordering as implemented in *ISAAC* [21]. For the derivation of sum-of-product expressions, both approaches are based on determinant expansion, follow the same order of expansion, and both use the cache to store the minors. But the DDD-based approach has the following several subtle differences from *ISAAC*. First, we impose the fixed ordering rule so that the structure of expansion is canonical. Further, the structure of expansion is formalized as a binary decision diagram, where each time only one matrix entry is considered. Then, the generation of sum-of-product expressions for a symbolic determinant is broken into three separate steps: 1) entry labeling (vertex ordering), 2) construction of the diagram with the chosen order, and 3) generation of the product terms from the diagram. These considerations enable us to exploit the understandings and implementation of BDD's developed mainly in the past decade in the area of formal verification and logic synthesis. We further show that symbolic manipulation on sum-of-product expressions can be performed much more efficiently on the proposed diagrams.

B. Comparison with Hierarchical Symbolic Analysis

DDD's can be viewed as a special form of sequences of expressions as used in hierarchical symbolic analysis [24], [40]. They differ in how they are created and the canonicity of the representation, which lead to several fundamental differences in their performance.

First, the DDD representation is unique. For a given circuit, regardless of which algorithms to use, the generated code based on the DDD representation must always be the same (should be able to be compared simply using UNIX shell command *diff*). The canonicity property may be useful for formal analog verification.

Second, symbolic manipulation with DDD's is simpler than with arbitrarily nested sequences of expressions. From the DDD's, the expanded sum-of-product expressions can be generated by a simple DDD traversal; the significant terms can be generated efficiently by finding k-shortest DDD paths; the s-expanded expressions can be derived in linear time in the size of a DDD [39]. Very recently, we have shown that the problem of deriving simplified, reliable, and interpretable symbolic network functions can be performed effectively and efficiently with DDD's [43]. In contrast, manipulation and approximation of arbitrarily nested symbolic expressions are known to be more difficult and involved [18], [27], [37].

Third, in comparison with numerical evaluation using hierarchical symbolic analyzer *SCAPP*, which is the compiled partial **LU** decomposition by Gaussian elimination [44], numerical evaluation with DDD's is division-free, manipulates fewer derived values, and generally adds/subtracts the values with less differing magnitudes. Furthermore, since pivoting is not employed in *SCAPP*, repetitive numerical evaluation with the generated code may be subject to the numerical accuracy problem. We note that compiled-code simulation has been studied in the area of numerical circuit simulation [48], and partial **LU** decomposition by Gaussian elimination itself has been exploited in the context of circuit tearing, for example as in [49]. If targeted at circuit simulation, **LU** decomposition by Gaussian elimination has been observed to be less preferred over the Crout method or other methods of **LU** decomposition [44].

Finally, it is worth noting that efficient linear(ized) circuit analysis can be accomplished via numerical reduced-order modeling techniques such as asymptotic waveform evaluation [34], the PVL algorithm [16], and the Arnoldi method [31]. It is very intriguing to combine DDD's and these techniques for possible **symbolic** reduced-order modeling.

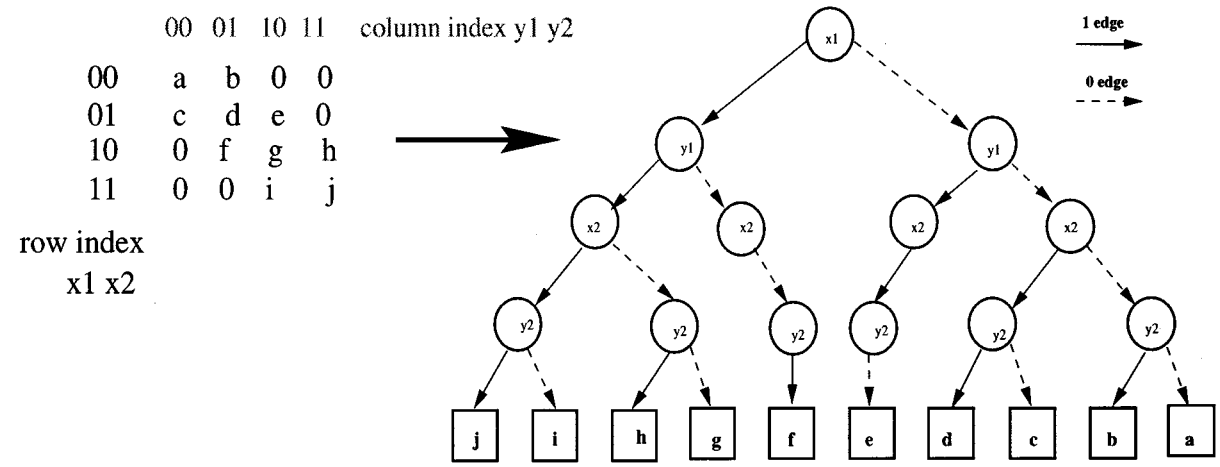


Fig. 17. An MTBDD representation of matrix $\boldsymbol{M}$.

C. Relevance to Other Decision Diagram Concepts

The notion of DDD's comes from the application of ZBDD's, a variant of BDD's, to represent symbolic matrix determinants. Each vertex in the DDD represents a symbolic determinant and is defined by the determinant expansion rule (2), where operations are the addition and multiplication in normal algebra. We note that several other extensions of BDD's for multivalued functions and arithmetic applications have been made; for example, multiterminal binary decision diagrams [11]–[13], [20] (also called algebraic decision diagrams [3]), hybrid decision diagrams [14], binary moment diagrams (BMD's) [6], and several others as described in the book edited by Sasao and Fujita [36]. Among them, Multiterminal BDD's (MTBDD's) have been explored for the implementation of matrix algebra.

MTBDD's is an extension of BDD's with multiple terminals, each of which as a real value [11]–[14], [20]. MTBDD's can be used to represent matrices by observing that the row and column indexes of a matrix can be encoded as Binary vectors, say $\{x_1x_2\cdots x_k\}$ and $\{y_1y_2\cdots y_l\}$ ($k=l$ for square matrices), and then the matrix can be conceptually viewed as a function f: $B^{k+l} \rightarrow R$ where R is the set of nonzero matrix entries. In the resulting MTBDD for representing this multivalued Boolean function f, each terminal represents a nonzero matrix entry, and each nonterminal vertex is labeled by either a row or a column encoding bit (x_i or y_i). Each path from the root to a terminal defines the row and column position at which the matrix entry—represented by the terminal—locates. For example, Fig. 17 shows an MTBDD representation of the matrix $\boldsymbol{M}$. As shown by Fujita, McGeer and Yang, using MTBDD's, many matrix algebraic operations such as Strassen matrix multiplication, spectral transforms, and **LU** decomposition can be performed elegantly [20]. As a representation of **matrices**, MTBDD's differ from DDD's, which is a representation of **matrix determinants**. Although it may be argued that a matrix determinant may be calculated explicitly or implicitly based on the determinant definition (1) in Section II-B from the MTBDD representation of the matrix and basic MTBDD operations described in [20]. However, the computational procedure and results would be significantly complicated, and MTBDD's have not been used for determinant computation. To the best of our knowledge, the introduction of DDD's represents the first effort in exploring BDD's and their variations for representing and manipulating matrix determinants and cofactors. On the other hand, DDD's represent matrix determinants, not matrices, and are not adequate for implementing general matrix algebra. Nevertheless, matrices and determinants are intriguingly related. It is interesting to explore more connections between MTBDD's and DDD's.

BMD's[6] are more closely connected to DDD's. BMD's provide a canonical representation for multilinear functions. They are a variation of BDD's where the expansion rule is the following decomposition rule of function f

$$f = f_{\overline{x}} + x(f_x - f_{\overline{x}})$$

where f_x (respectively, $f_{\overline{x}}$) denotes the positive (respectively, negative) cofactor of f with respect to x, i.e., the function resulting when constant one (respectively, zero) is substituted for x. Note that a determinant is a multilinear function in its entries. Recall the DDD expansion rule (2)

$$\det(\boldsymbol{A}) = a_{r,c}(-1)^{r+c}\det(\boldsymbol{A}_{a_{r,c}}) + \det(\boldsymbol{A}_{\overline{a}_{r,c}}).$$

Here, $\det(\boldsymbol{A}_{\overline{a}_{r,c}})$ is the remainder of $\det(\boldsymbol{A})$, which is equal to $\det(\boldsymbol{A})|_{a_{r,c}=0}$, i.e., the value of $\det(\boldsymbol{A})$ by substituting $a_{r,c}$ by zero. It can be verified that

$$(-1)^{r+c}\det(\boldsymbol{A}_{a_{r,c}}) = \det(\boldsymbol{A})|_{a_{r,c}=1} - \det(\boldsymbol{A})|_{a_{r,c}=0}.$$

Therefore, the DDD representation can be viewed as a special case of the BMD representation.

However, the matrix determinants as a special form of multilinear functions have several special properties. For example, the sign can be determined nicely using the sign rule (Section III) and be attached as part of a vertex, whereas in BMD's, signs are encoded as edge weights. The BMDs' objective is for formal verification of arithmetic digital circuits, where matrix

[6]The connection of BMD's to ZBDD's and MTBDD's has been noted by several researchers [6], [14], [30].

operations such as multiplication are emphasized, whereas in our application, solving a system of linear equations is the objective and the representation and manipulation of symbolic determinants and cofactors of **sparse** matrices are of primary interest. Nevertheless, many ideas developed in the area of decision diagrams such as using multiterminals and attributed edges to represent polynomials and numeric coefficients can be adapted to enhance the power of DDD's.

VIII. Conclusion

In this paper, a new graph representation, called DDD's, for symbolic matrix determinants is introduced and symbolic analysis algorithms for analog circuits are presented. Unlike previous approaches based on either the expanded form or the nested form representations of symbolic expressions, DDD-based symbolic analysis exploits the sparsity and sharing in a canonical manner. We described an efficient vertex-ordering heuristic and proved that it is optimum for ladder-structured circuits; in this case, the number of DDD vertices is equal to the number of nonzero matrix entries. We emphasize that the DDD size depends on the size of a circuit, its structure and sparsity, as well as the chosen vertex ordering. In the worst case, the DDD size can grow exponentially with the size of a circuit. Fortunately, for practical circuits, we have observed that with the proposed vertex-ordering heuristic, the numbers of DDD vertices are quite small—usually several orders of magnitude smaller than the numbers of product terms in the expanded form. Generating the complete sum-of-product symbolic expressions from the DDD representation offers orders-of-magnitude improvement in both CPU time and memory usages over symbolic analyzers *ISAAC* and *Maple-V* for large analog circuits. It enables the exact and canonical symbolic analysis of such large analog circuits as μA741 for the first time.

We also compared the DDD-based approach with the state-of-art hierarchical symbolic analyzer *SCAPP* in generating the sequences of expressions for network transfer functions. For ladder-structured circuits, the DDD-based approach uses only two thirds of multiplications as required by *SCAPP*. For large circuits that do not have ladder-like structures such as μA741 opamp, the sequences of expressions generated by the DDD-based approach are generally manageable by modern computers, but can be substantially longer than those from *SCAPP*. However, the DDD representation is unique. That is, for a given circuit, regardless of which algorithms to use, the generated code based on the DDD representation must always be the same (should be able to be compared simply using UNIX shell command *diff*). The canonicity property may be useful for formal analog verification.

In contrast to the nested form as used in *SCAPP*, the DDD representation is more amenable to efficient symbolic manipulation. As shown in this paper, symbolic analysis algorithms such as driving cofactor computation, network function construction, sensitivity calculation, and generating significant terms, can be performed in time almost linear in the number of DDD vertices. Very recently, we have shown other important manipulations such as deriving s-expanded symbolic expressions and symbolic approximation can be accomplished in a similar manner [39], [43].

In comparison with numerical **LU** decomposition as in *SPICE* and symbolic **LU** decomposition as in *SCAPP*, DDD evaluation is division-free, manipulates fewer derived values, and generally adds/subtracts the values with less differing magnitudes. Inspired by these features, as well as the canonicity and compactness of the DDD representation, research is being extended to exploit the full potential of canonical symbolic analysis for the design and test automation of analog circuits, as well as in general symbolic algebra.

Acknowledgment

The authors would like to thank Prof. F. Brewer and his research group at the University of California, Santa Barbara, for providing them with their *HomeBrew* BDD package, which accelerated significantly their initial DDD implementation, Prof. G. Gielen of Katholieke Universiteit Leuven, Belgium, and Prof. M. Hassoun of Iowa State University, Ames, for providing them their symbolic circuit analysis packages *ISAAC* and *SCAPP*. They are also grateful to Prof. T. Sasao of Kyushu Institute of Technology, Japan, and the anonymous reviewers for valuable comments that improve the presentation of this paper.

References

[1] S. B. Akers, "Binary decision diagrams," *IEEE Trans. Comput.*, vol. C-27, pp. 509–516, June 1976.

[2] G. E. Alderson and P. M. Lin, "Computer generation of symbolic network functions—a new theory and implementation," *IEEE Trans. Circuit Theory*, vol. CT-20, pp. 48–56, Jan. 1973.

[3] R. I. Bahar, E. A. Frohm, C. M. Gaona, G. A. Hachtel, E. Macii, A. Pardo, and F. Somenzi, "Algebraic decision diagrams and their applications," in *Proc. IEEE/ACM Int. Conf. Computer-Aided Design*, Nov. 1993, pp. 188–191.

[4] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Trans. Comput.*, vol. C-37, pp. 677–691, Aug. 1986.

[5] R. E. Bryant, "Binary decision diagrams and beyond: Enabling technologies for formal verification," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 1995, pp. 236–243.

[6] R. E. Bryant and Y. A. Chen, "Verification of arithmetic functions with binary moment diagrams," in *Proc. 32nd IEEE/ACM Design Automation Conf.*, San Francisco, CA, June 1995, pp. 535–541.

[7] K. S. Brace, R. L. Rudell, and R. E. Bryant, "Efficient implementation of a BDD package," in *Proc. 27th IEEE/ACM Design Automation Conf.*, June 1990, pp. 40–45.

[8] J. G. Broida and S. G. Williamson, *A Comprehensive Introduction to Linear Algebra*. Reading, MA: Addison-Wesley, 1989.

[9] R. Carmassi, M. Catelani, G. Iuculano, A. Liberatore, S. Manetti, and S. Marini, "Analog network testability measurement: A symbolic formulation approach," *IEEE Trans. Instrum. Meas.*, vol. 40, pp. 930–935, Dec. 1991.

[10] B. W. Char, *et al.*, *Maple V: Language Reference Manual*. Berlin, Germany: Springer-Verlag, 1991.

[11] E. M. Clarke, X. Zhao, M. Fujita, Y. Matsunga, P. C. McGeer, and J. Yang, "Fast Walsh transform computation using binary decision diagrams," presented at the IFIP WG 10.5 Workshop on Applications of Reed–Muller Expansion in Circuit Design, 1993.

[12] E. M. Clarke, M. Fujita, P. C. McGeer, J. Yang , and X. Zhao, "Multiterminal binary decision diagrams: An efficient data structure for matrix representation," presented at the Int. Workshop on Logic Synthesis (IWLS), Tahoe City, CA, May 23–26, 1993.

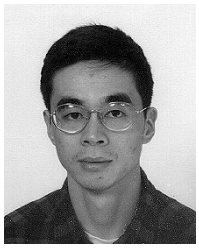
[13] E. M. Clarke, M. Fujita, P. C. McGeer, X. Zhao , and J. Yang, "Fast spectrum computation for logic functions using binary decision diagrams," in *Proc. IEEE Int. Symp. Circuits and Systems*, May 1994, pp. 275–278.

[14] E. M. Clarke, M. Fujita , and X. Zhao, "Multi-terminal binary decision diagrams and hybrid decision diagrams," in *Representations of Discrete Functions*, T. Sasao and M. Fujita, Eds. Norwell, MA: Kluwer Academic, 1996, pp. 93–108.
[15] T. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. Cambridge, MA: MIT Press, 1990.
[16] P. Feldmann and R. Freund, "Efficient linear circuit analysis by Pade approximation via the Lanczos process," *IEEE Trans. Computer-Aided Design*, vol. 14, pp. 639–649, May 1995.
[17] F. V. Fernández, J. D. Martín, A. Rodríguez-Vázquez, and J. L. Huertas, "On simplification techniques for symbolic analysis of analog integrated circuits," in *Proc. IEEE Int. Symp. Circuits and Systems*, 1992, pp. 1149–1152.
[18] F. V. Fernández, A. Rodríguez-Vázquez, J. D. Martín, and J. L. Huertas, "Accurate simplification of large symbolic formulae," in *Proc. IEEE Int. Conf. Computer Aided Design (ICCAD)*, 1992, pp. 318–321.
[19] F. V. Fernández and A. Rodríguez-Vázquez, "Symbolic analysis tools—the state of the art," in *Proc. IEEE Int. Symp. Circuits and System*, 1996, pp. 798–801.
[20] M. Fujita, P. C. McGeer, and J. C.-Y. Yang, "Multi-terminal binary decision diagrams: An efficient data structure for matrix representation," *Formal Meth. Syst. Design*, vol. 10, pp. 149–169, 1997.
[21] G. Gielen and W. Sansen, *Symbolic Analysis for Automated Design of Analog Integrated Circuits*. Norwell, MA: Kluwer Academic, 1991.
[22] G. Gielen, P. Wambacq, and W. Sansen, "Symbolic analysis methods and applications for analog circuits: A tutorial overview," *Proc. IEEE*, vol. 82, no. 2, pp. 287–304, Feb. 1994.
[23] M. M. Hassoun and P. M. Lin, "A new network approach to symbolic simulation of large-scale network," in *Proc. IEEE Int. Symp. Circuits and Systems*, 1989, pp. 806–809.
[24] ——, "A hierarchical network approach to symbolic analysis of large scale networks," *IEEE Trans. Circuits Syst.*, vol. 42, pp. 201–211, Apr. 1995.
[25] J.-J. Hsu and C. Sechen, "DC small-signal symbolic analysis of large analog integrated circuits," *IEEE Trans. Circuits Syst.*, vol. 41, pp. 817–828, Dec. 1994.
[26] P. M. Lin, *Symbolic Network Analysis*. Amsterdam, the Netherlands: Elsevier Science, 1991.
[27] ——, "Sensitivity analysis of large linear networks using symbolic program," in *Proc. IEEE Int. Symp. Circuits and Systems*, 1992, pp. 1145–1148.
[28] W. J. McCalla and D. O. Pederson , "Elements of computer-aided analysis," *IEEE Trans. Circuit Theory* , vol. CT-18 , pp. 14–26, Jan. 1971.
[29] S. Minato, "Zero-suppressed BDD's for set manipulation in combinatorial problems," in *Proc. 30th IEEE/ACM Design Automation Conf.*, Dallas, TX, June 1993, pp. 272–277.
[30] ——, *Binary Decision Diagrams and Applications for VLSI CAD*. Norwell, MA: Kluwer Academic, 1996.
[31] L. Miguel-Silveira, M. Kamon, I. Elfadel, and J. White, "A coordinate-transformed Arnoldi algorithm for generating guaranteed stable reduced-order models of RLC circuits," in *Proc. IEEE/ACM Int. Conf. Computer-Aided Design*, 1996, pp. 288–294.
[32] L. W. Nagel, "SPICE2: A computer program to simulate semiconductor circuits," Ph.D. dissertation, Univ. California, Berkeley, CA, May 1975.
[33] A. R. Newton and A. L. Sangiovanni-Vincentelli, "Relaxation-based electrical simulation," *IEEE Trans. Computer-Aided Design*, vol. CAD-3, pp. 308–331, Oct. 1984.
[34] L. T. Pillage and R. A. Rohrer, "Asymptotic waveform evaluation for timing analysis," *IEEE Trans. Computer-Aided Design*, vol. 9, pp. 352–366, Apr. 1990.
[35] P. Sannuti and N. N. Puri, "Symbolic network analysis—an algebraic formulation," *IEEE Trans. Circuits Syst.*, vol. CAS-27, pp. 679–687, Aug. 1980.
[36] T. Sasao and M. Fujita, *Representations of Discrete Functions*. Norwell, MA: Kluwer Academic, 1996.
[37] S. J. Seda, M. G. R. Degrauwe, and W. Fichtner, "Lazy-expansion symbolic expression approximation in SYNAP," in *Proc. IEEE Int. Conf. Computer Aided Design (ICCAD)*, 1992, pp. 310–317.
[38] C.-J. Shi and X. Tan, "Symbolic analysis of large analog circuits with determinant decision diagrams," in *Proc. IEEE/ACM Int. Conf. Computer-Aided Design*, San Jose, CA, Nov. 1997, pp. 366–373.
[39] ——, "Efficient derivation of exact s-expanded symbolic expressions for behavioral modeling of analog circuits," in *Proc. IEEE Custom Integrated Circuits Conf. (CICC'98)*, San Clara, CA, May 1998, pp. 463–466.
[40] J. A. Starzky and A. Konczykowska, "Flowgraph analysis of large electronic networks," *IEEE Trans. Circuits Syst.*, vol. CAS-33, pp. 302–315, Mar. 1986.
[41] X. Tan and C.-J. Shi, "Hierarchical symbolic analysis of large analog circuits with determinant decision diagrams," in *Proc. IEEE Int. Symp. Circuits and Systems*, vol. VI, May 1998, pp. 318–321.
[42] ——, "Balanced multilevel multiway partitioning of large analog circuits for hierarchical symbolic analysis," in *Proc. Asia and South Pacific Design Automation Conf. (ASP-DAC)*, Jan. 18–21, 1999, pp. 1–4.
[43] ——, "Interpretable symbolic small-signal characterization of large analog circuits using determinant decision diagrams," in *Proc. Design, Automation, and Test in Europe Conf. and Exhibition (DATE)*, Mar. 10–13, 1999, pp. 448–453.
[44] J. Vlach and K. Singhal, *Computer Methods for Circuit Analysis and Design*, New York: Van Nostrand Reinhold, 1994.
[45] P. Wambacq, G. Gielen , and W. Sansen, "A cancellation-free algorithm for the symbolic simulation of large analog circuits," in *Proc. IEEE Int. Symp. Circuits and Systems*, 1992, pp. 1157–1160.
[46] P. Wambacq, *et al.*, "A family of matroid intersection algorithms for the computation of approximate symbolic network functions," in *Proc. IEEE Int. Symp. Circuits and Systems*, 1996, pp. 806–809.
[47] P. Wambacq, G. Gielen , and W. Sansen, "A new reliable approximation method for expanded symbolic network functions," in *Proc. IEEE Int. Symp. Circuits and Systems*, 1996, pp. 584–587.
[48] W. T. Weeks, A. J. Jimenez, G. W. Mahoney, D. Mehta, H. Wassemzadeh, and T. R. Scott, "Algorithms for ASTAP—A network analysis program," *IEEE Trans. Circuit Theory*, vol. CT-20, pp. 628–634, Nov. 1973.
[49] P. Yang, "An investigation of ordering, tearing, latency algorithms for the time-domain simulation of large circuits," Ph.D. dissertation, Univ. Illinois at Urbana-Champaign, Urbana, IL, 1980.
[50] Z. You, E. Sánchez-Sinencio, and J. P. de Gyvez, "Analog system-level fault diagnosis based on a symbolic method in the frequency domain," *IEEE Trans. Instrum. Meas.*, vol. 44, no. 1, pp. 28–35, Jan. 1995.
[51] Q. Yu and C. Sechen, "A unified approach to the approximate symbolic analysis of large analog integrated circuits," *IEEE Trans. Circuits Syst.*, vol. 43, pp. 656–669, Aug. 1996.
[52] ——, "Efficient approximation of symbolic functions using matroid intersection algorithms," *IEEE Trans. Computer-Aided Design*, vol. 16, pp. 1073–1081, Oct. 1997.

C.-J. Richard Shi (M'91–SM'99) received the B.S. and M.S. degrees in electrical engineering from Fudan University, Shanghai, China, in 1985 and 1988, respectively, the M.A.Sc. degree in electrical engineering and the Ph.D. degree in computer science from the University of Waterloo, Waterloo, Ont., Canada, in 1991 and 1994, respectively.

He is currently an Assistant Professor in the Department of Electrical Engineering, University of Washington, Seattle. His research interests include methodologies and tools for systems-on-a-chip design, with the particular emphasis on analog, mixed-signal, and deep-submicron design and test automation. He has published more than 70 technical papers, and has been a principal investigator of more than $2M in research funding from DARPA, NSF, USAF, CDADIC, and industry since 1995. He is a consultant to several semiconductor and EDA companies.

Dr. Shi co-founded IEEE/ACM/VIUF International Workshop on Behavioral Modeling and Simulation, and served as its Technical Program Chair from 1997 to 1999. Having been involved in IEEE DASC 1076.1 VHDL-AMS Working Group since 1994, he is one of the contributors to, and promoters of, IEEE std 1076.1-1999 standard language (VHDL-AMS) for the description and simulation of mixed-signal/mixed-technology systems. He has delivered tutorials on VHDL-AMS and behavioral modeling at several conferences including DAC, EuroDAC, and ASP-DAC. He has been a recipient or co-recipient of several awards including the T. D. Lee Physics Award for excellence in graduate study from Fudan, University of Waterloo Outstanding Achievement in Graduate Studies Award, the Natural Sciences and Engineering Research Council of Canada Doctoral Prize, a National Science Foundation CAREER Award, four Best Paper Awards (including the 1999 IEEE/ACM Design Automation Conference Best Paper Award and the 1999 IEEE VLSI Test Symposium Best Paper Award), and three other Best Paper Award Nominations (ASP-DAC'98, EuroDAC'96, and ASP-DAC'95). He is a member of IEEE Design Automation Standards Committee. He is an Associate Editor of IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS–II.



Xiang-Dong Tan (S'96–M'99) received the B.S. and M.S. degrees in electrical engineering from Fudan University, Shanghai, China, in 1992 and 1995, respectively, and the Ph.D. degree in electrical and computer engineering from the University of Iowa, Iowa City, in 1999.

He is currently a Member of Technical Staff at Monterey Design Systems, Monterey, CA. He worked with Rockwell Semiconductor Systems in the summer of 1997, and Avant! Corporation in the summer of 1998. He was a Research Assistant in the Department of Electrical Engineering, University of Washington, Seattle, from September 1998 to April 1999. His current research interests include very large scale integration (VLSI) physical design automation, symbolic analysis of large analog circuits, layout optimization for performance, timing, power, and clock tree synthesis.

Dr. Tan received a Best Paper Award from the 1999 IEEE/ACM Design Automation Conference in 1999 and the First-Place Student Poster Award from the 1999 Spring Meeting of the Center for Design of Analog Digital Integrated Circuits (CDADIC). He received a Best Graduate Award in 1992 and a number of Excellent College Student Scholarships from 1988–1992, all from Fudan University.