Efficient Approximation of Symbolic Expressions for Analog Behavioral Modeling and Analysis

Sheldon X.-D. Tan, *Member, IEEE*, C.-J. Richard Shi, *Senior Member, IEEE*

***Abstract*— Efficient algorithms are presented to generate approximate expressions for transfer functions and characteristics of large linear analog circuits. The algorithms are based on a compact determinant decision diagram (DDD) representation of exact transfer functions and characteristics. Several theoretical properties of DDDs are characterized, and three algorithms, namely, based on dynamic programming, based on consecutive k-shortest path based, and based on incremental k-shortest path, are presented in this paper. We show theoretically that all three algorithms have time complexity linearly proportional to $|DDD|$, the number of vertices of a DDD, and that the incremental k-shortest path based algorithm is fastest and the most flexible one. Experimental results confirm that the proposed algorithms are the most efficient ones reported so far, and are capable of generating thousands of dominant terms for typical analog blocks in CPU seconds on a modern computer workstation.**

***Index Terms*— analog symbolic analysis, circuit simulation, determinant decision diagrams, matrix determinant, behavioral modeling**

I. Introduction

As more custom very large scale integrated (VLSI) circuit designs become system-on-a-chip (SoC) designs, efficient implementations of analog/radio frequency (RF) building blocks in SoC systems become increasingly important. But design automation techniques for analog circuits are significantly lagging behind their counterparts for digital circuits. One of the challenging problems in analog design is that analog circuits typically have many characteristics, and they depend in a very complicated way on circuit, layout process and environment parameters.

In this paper, we consider the problem of generating simple yet accurate symbolic representation of circuit transfer function and characteristics in terms of circuit parameters for linear(ized) analog integrated circuits. It is well known that circuit transfer function and characteristics are dominated by a small number of product terms called *significant* or *dominant*. For instance, Fig. 1 shows a CMOS two-stage opamp circuit. The simplified MOS small-signal model is shown in Fig. 2. If we ignore c_{bg}, c_{sb} and g_{mb} (as they are typically small compared to other parameters) in the MOS model and treat the M_5 as an ideal current source and M_7 as a resistor, the exact transfer function from input v_{in2} to v_{out} (v_{in1} is shunted to ground), as shown in Appendix , contains 16 product terms in the numerator and 60 terms in the denominator. We can see that each coefficient expression of s^k is dominated by a few product terms. For example, for the coefficient of s^1 in the denominator, the first product term amounts to 86% of the total magnitude of the coefficient and the first two terms amount to 97% of the total magnitude. If these errors are acceptable, the remaining terms can be neglected. With 5% error, the transfer function can be simplified as shown in Eq. (1).

$$H(s) = \frac{v_{out}}{v_{in2}} = \frac{g_{m2}g_{m6} + s^1(g_{m2}CC)}{(\frac{1}{r_{o2}} + \frac{1}{r_{o4}})(\frac{1}{r_{o6}} + \frac{1}{r_{o7}}) - s^1(g_{m6}C_C) + s^2(c_{db2} + c_{db4} + c_{gs6} + C_L)C_C} \quad (1)$$

Notice that the symbolic approximation are typically carried out around some nominal numerical values (points) of the devices involved, the generated model will approximate actual devices well when the sized device parameters are close to the nominal values used for model generation. However, when using a staged optimization approach, the nominal points will move and model will be updated adaptively. As a result, the optimized devices will be very close to the nominal points of the model generated in the final stage (as the model is re-centered after every stage). Such nominal point approximation strategy used in this work is also adopted by most of symbolic approximation methods [1]–[7]. During the approximation, we monitor both magnitudes and phases of the transfer functions until errors are within the user-specified error bounds for the frequency range.

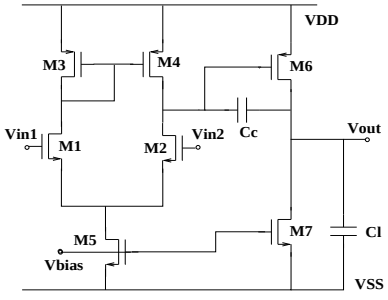

Fig. 1. A simplified two-stage CMOS opamp [8].

As illustrated in [9], simple yet accurate symbolic expressions can be interpretable by analog designers to gain insight into the circuit behavior, performance and stability, and are important for many applications in circuit design such as transistor sizing and optimization, topology selection, analysis, behavioral modeling, fault simulation,

Manuscript received Oct 10, 2002; revised May 20, 2003.

The work of Sheldon X.-D. Tan was supported by UC Senate Resaerch Funds. The paper was recommended by Associate Editor Rob A. Rutenbar.

Sheldon X.-D. Tan is with Department of Electrical Engineering, University of California, Riverside, Riverside, CA 92521 USA (e-mail: stan@ee.ucr.edu).

C.-J. Richard Shi is with Department of Electrical Engineering, University of Washington, Seattle, WA 98195 USA.

Publisher Item Identifier TCAD.2004.3000

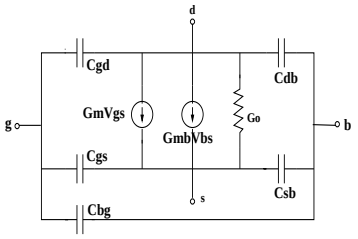

Fig. 2. MOSFET small signal model.

testability analysis and yield enhancement [10]. Efficient symbolic techniques for linear(ized) analog circuit analysis are the basis of distortion analysis of weakly nonlinear circuits [11], [12], symbolic modeling of time-varying systems such as mixers [13].

Previous attempts to generate interpretable expressions use various symbolic analysis methods to generate sum-of-product representations for network functions. This area has been studied extensively in 1960s-1980s [14]. The resulting approaches, however, are only feasible for very small circuits, since the number of expanded product terms grows exponentially with the size of a circuit, and the resulting expressions are no longer interpretable by analog designers. Recently, various approximation schemes have been developed. Approximation after generation is reliable but it requires the expansion of product terms first [6], [9], [15]. Some improvements based on nested expressions have been proposed [16], [17]. But they generally suffer symbolic term-cancellation and align-term problems. Approximation during generation extracts only significant product terms [2], [5], [7]. It is very fast, but has two major deficiencies: First, if accurate expressions are needed, the complexity of the approach becomes exponential. Second, it works only for transfer functions. Other small-signal characteristics such as sensitivities, symbolic poles and zeros, cannot be extracted in general. At the same time, several approximation before generation techniques [7], [18] were proposed in which the complexity of a circuit is simplified before symbolic analysis methods are applied. Recently, a signal-flow graph based approximation before generation method was proposed [19] and demonstrated successfully to symbolic pole and zero generation. Symbolic analysis based on the concept of signal paths in control theory was employed for pole and zero location analysis [3], [20].

In this paper, we present efficient algorithms of generating dominant terms for deriving interpretable symbolic expressions based on a compact determinant decision diagram (DDD) representation of circuit transfer functions [21], [22]. We show that dominant term generation can be performed elegantly by DDD graph manipulations. Since we start with exact symbolic expressions in DDD representations, our new approximation methods feature both the reliability in the approximation-after-generation methods and the capability in approximation-before/after-generation for analyzing large analog circuits. Experimental results show that our algorithms outperform the best dominant term generation method reported so far [4], [23].

Some preliminary results of this paper were presented in [24], [25]. This paper is organized as follows. Section II reviews the concepts of DDDs and s-expanded DDDs. Section III presents a dynamic programming based term generation algorithm based on the work of [4], [23]. Section IV describes the consecutive k-shortest path generation algorithm. Section V presents the most efficient algorithms based on incremental k-shortest paths. Experimental results are presented in Section VI. Section VII concludes the paper.

II. DDDs and s-Expanded Coefficient DDDs

In this section, we provide a brief overview of the notion of determinant decision diagrams [21]. We review how a multiple-root DDD can be used to represent the symbolic coefficients of an s polynomial [22].

Determinant Decision Diagrams [21] are compact and canonical graph-based representation of determinants. The concept is best illustrated using a simple RC filter circuit shown in Fig. 3. Its system equations can be written as

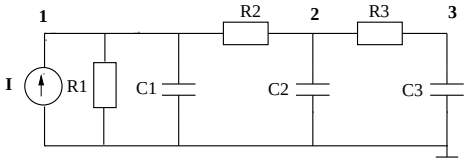

Fig. 3. An example circuit.

$$\begin{bmatrix} \frac{1}{R_1}+sC_1+\frac{1}{R_2} & -\frac{1}{R_2} & 0 \\ -\frac{1}{R_2} & \frac{1}{R_2}+sC_2+\frac{1}{R_3} & -\frac{1}{R_3} \\ 0 & -\frac{1}{R_3} & \frac{1}{R_3}+sC_3 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} I \\ 0 \\ 0 \end{bmatrix}$$

We view each entry in the circuit matrix as one distinct symbol, and rewrite its system determinant in the left-hand side of Fig. 4. Then its DDD representation is shown in the right-hand side.

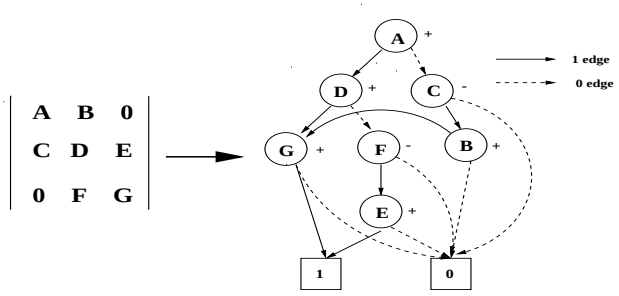

Fig. 4. A matrix determinant and its DDD representation.

A DDD is a signed, rooted, ordered, directed acyclic graph with two terminal vertices, namely the *0-terminal* vertex and the *1-terminal* vertex. Each non-terminal DDD vertex is labeled by a symbol in the determinant denoted by a_i (A to G in Fig. 4), and a positive or negative sign denoted by $s(a_i)$. It *originates* two outgoing edges, called *1-edge* and *0-edge*. Each vertex a_i represents a symbolic expression $D(a_i)$ defined recursively as follows: $D(a_i) = a_i\ s(a_i)\ D_{a_i} + D_{\overline{a}_i}$, where D_{a_i} and $D_{\overline{a}_i}$ represent, respectively, the symbolic expressions of the vertices pointed by the 1-edge and 0-edge of a_i. The 1-terminal vertex represents expression 1, whereas the 0-terminal

vertex represents expression 0. For example, vertex E in Fig. 4 represents expression E, and vertex F represents expression $-EF$, and vertex D represents expression $DG-FE$. We also say that a DDD vertex D represents an expression defined by the DDD subgraph rooted at D.

A *1-path* in a DDD corresponds a product term in the original DDD, which is defined as a path from the root vertex (A in our example) to the 1-terminal including all symbolic symbols and signs of the vertices that originate all the 1-edges along the 1-path. In our example, there exist three 1-paths representing three product terms: ADG, $-AFE$ and $-CBG$. The root vertex represents the sum of these product terms. Size of a DDD is the number of DDD vertices, denoted by $|DDD|$.

DDD graph is also an ordered graph like Binary Decision Diagrams (BDDs). This implies that the order of each symbol in any 1-path from (root vertex to 1-terminal) is fixed with respect to other symbols. The ordering used in our example is $A > C > B > D > F > E > G$. Notice that the size of a DDD depends on the size of a circuit in a complicated way. Both circuit topology and vertex ordering have huge impacts on the DDD sizes. Given the best vertex ordering, if the underlying circuit is a ladder circuit, $|DDD|$ is a linear function of the sizes of the circuit. For general circuits, the size of DDD graph may grow exponentially in the worse case. But like BDDs, with proper vertex ordering, the DDD representations are very compact for many real circuits [21], [22].

Given a symbolic matrix where each entry is a distinct symbol, and a fixed order of each label appears in the DDD starting from its root. Then the expansion of matrix can be carried out with respect to a non-zero element:

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

A DDD vertex pointed by 0-edge, which comes from vertex $a_{r,c}$, represents a determinant obtained by setting $a_{r,c} = 0$ and is denoted as $det(\mathbf{A}_{\overline{a}_{r,c}})$. The corresponding portion of the DDD graph for one-step determinant expansion of Eq. (2) is shown in Fig. 5.

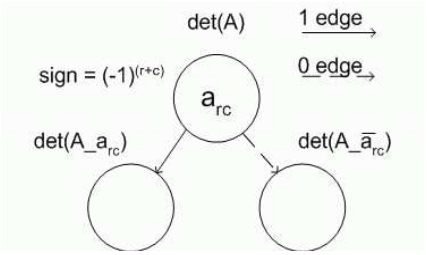

Fig. 5. A determinant expansion and its DDD representation.

It can be seen that the multiplication operation is represented by a 1-edge and the addition operation is represented by a 0-edge. The sign of each vertex is the sign used when the vertex is used to develop the determinant. If a DDD is constructed by using Laplace development, all the DDD vertices linked by 0-edges will have the same row index or column index. If the vertex ordering that we use to develop a determinant is fixed, there exists a *unique DDD representation* of the determinant for that vertex ordering. This is the DDD canonical property.

To exploit the DDD to derive circuit characteristics, we need to directly represent circuit parameters not matrix entries. To this end, s-expanded DDDs are introduced [22]. Consider again the circuit in Fig. 3 and its system determinant. Let us introduce a unique symbol for each circuit parameter in its admittance form. Specifically, we introduce $a = \frac{1}{R_1}$, $b = f = \frac{1}{R_2}$, $d = e = -\frac{1}{R_2}$ $g = k = \frac{1}{R_3}$, $i = j = -\frac{1}{R_3}$, $C_1 = c, h = C_2, l = C_3$. Then the circuit matrix can be rewritten as

$$\begin{bmatrix} a+b+cs & d & 0 \\ e & f+g+hs & i \\ 0 & j & k+ls \end{bmatrix}$$

The original 3 product terms will be expanded to 23 product terms in different powers of s. We can represent these product terms nicely using a slight extension of the original DDD, as shown in Fig. 6. This DDD has exactly the same properties as the original DDD except that there are four roots representing coefficients of s^0, s^1, s^2, s^3. Each DDD root represents a symbolic expression of a coefficient in the corresponding s polynomial. Each such DDD is called a *coefficient* DDD, and the resulting DDD is a *multiple-root* DDD. The original DDD in which s is contained in some vertices is called *complex* DDD. The s-expanded DDD can be constructed from the complex DDD in linear time in the size of the original complex DDD [22], [26].

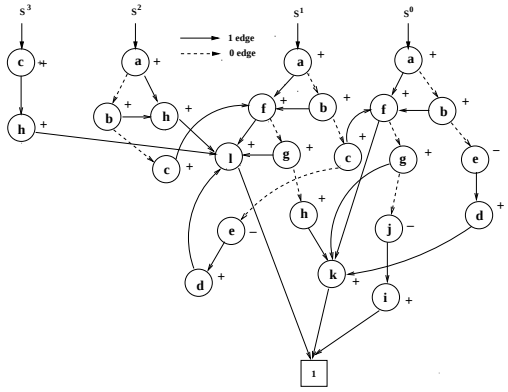

Fig. 6. An s-expanded DDD.

Before we generate the dominant terms, one problem we need to consider is symbolic cancellation. Symbolic canceling terms arise from the use of the MNA formulation in analog circuits. For instance, consider the s-expanded DDD in Fig. 6. Since $g = k = \frac{1}{R_3}$ and $i = j = -\frac{1}{R_3}$, term $agks^0$ cancels term $-ajis^0$ in the coefficient DDD of s^0. Our experiments show that 70-90% terms are canceling terms. Clearly it is inefficient to generate the 70%-90% terms that will not show up in the final expressions.

For this example the resulting cancellation-free DDD is shown in Fig. 7.

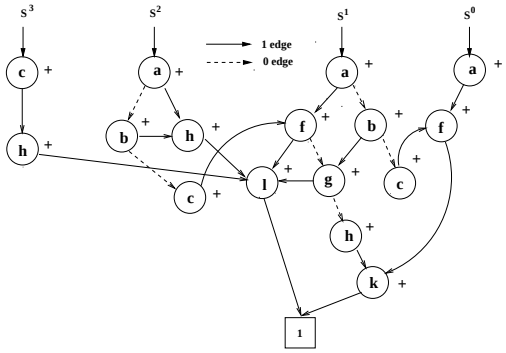

Fig. 7. The cancellation-free multi-root DDD.

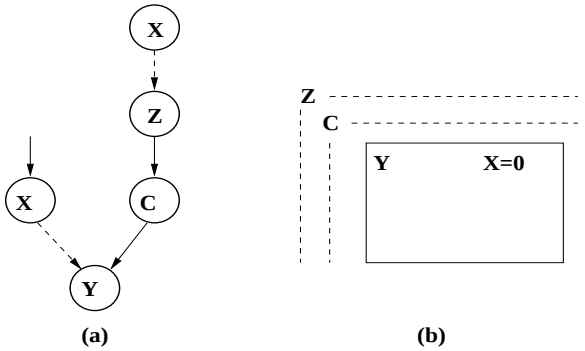

Fig. 8. A DDD vertex with both incoming 1-edge and 0-edge

III. Dynamic Programming based Generation of Dominant Terms

In this section, we present a dynamic programming (DP) based algorithm for generating dominant terms using DDDs. The original idea of using dynamic programming was proposed by Verhaegen and Gielen [4], [23], but was implemented based on non-canonical DDD graphs with node duplication. Our implementation is based on canonical DDD graphs without dynamic removal of canceling terms, and therefore is faster than the one in [4], [23].

To derive the DP algorithm, we characterize here some theoretical properties of DDDs. First, we note that the vertex ordering heuristic used to construct DDDs is based on Laplace expansion of a circuit matrix along the row or column [21]. We also know from the canonical property of DDD that the structure of a DDD is unique under a fixed vertex order, i.e., independent of how it is constructed. We thus have the following lemma.

Lemma 1: All 0-edge linked vertices come from either the same row or the same column of the original circuit matrix.
Proof: We know that a complex DDD is constructed by using Laplace development. To ease our proof, we repeat the determinant expansion rule (2) in the following:

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

In a DDD graph, the *addition* relationship between cofactor $a_{r,c}(-1)^{r+c} det(\mathbf{A}_{a_{r,c}})$ and remainder $det(\mathbf{A}_{\overline{a}_{r,c}})$ is represented by a 0-edge between two DDD vertices representing the two expressions as shown in Fig. 5. Since all the $a_{r,c}$ are selected from a row or a column by Laplace development rule, so all the 0-edge linked DDD vertices will have the same row index or column index. Further the row or column must exist in the remainder $det(\mathbf{A}_{\overline{a}_{r,c}})$.

With Lemma 1 as the basis, we can show the following main result.

Theorem 1: The incoming edges of a non-terminal vertex in a complex DDD or native s-expanded DDD are either all 0-edges or all 1-edges.

Proof: Suppose there exists a DDD vertex Y that has an incoming 0-edge from DDD vertex X, and an incoming 1-edge from vertex C as shown in Fig. 8(a). The DDD subgraph rooted at Y therefore represents a remainder by the development of X. According to the determinant development rule (2), the 0-edge between X and Y means that both the row and column of X exist in the remainder represented by Y (actually, X and Y are in the same row or column according to Lemma 1). Also the element of X in the remainder Y will become zero as shown in Fig. 8(b).

On the other hand, Y and C are not in the same row and column as Y is obtained by Laplace development with respect to C. Since element X is zero, X will appear before C with a 0-edge in the path (or there could be a number of elements in between linked with 0-edges; as they must come from the same row or column). Let the first non 0-edge linked vertex to be Z (Z can be C). The fact that Z is 1-edge linked implies the minor after Z is developed will not have the row and column of Z. As a result, X would have been crossed out since it is in the same row or column as Z. This contrasts with the fact that X appears in the remainder Y.

This lemma also holds for *native* s-expanded DDDs by noticing that 0-edges between DDD vertices may also represent the relationship among the different symbolic circuit parameters in a non-zero element in a determinant. *Native* here means that the s-expanded DDDs are obtained from complex DDDs without any structure modification.

Now we are ready to describe a DDD-based DP algorithm for generating k dominant terms. From Lemma 1 and Theorem 1, we know that we only need to calculate dominant terms for 1-edge pointed DDD vertices. Let D be a 1-edge pointed vertex. We use *D.counter* to keep track of the number of dominant terms generated for the vertex D (D is included in the terms). We use an ordered array, denoted as *D.term-list*, to keep track of those generated dominant terms in the minor represented by D, where *D.term-list[1]*, *D.term-list[2]*,... represents the largest term (first dominant term), the second largest term (second dominant term). *D.counter* is initially set to 1 for all 1-edge pointed vertices, and can be increased up to k.

As shown in pseudo code in Fig. 9, to find the k dominant terms at a 1-edge pointed vertex D, we first check if such k terms already exist in the *term-list* in GetKDomiTerms(D, k). If they do not exist, they will be generated by invoking ComputeKDominantTerm(D, k). In ComputeKDominantTerm(D, k), the largest term is computed and stored in the *term-list* of D by visiting all the 0-edge linked DDD vertices. Each time a dominant term is computed, the corresponding vertex $V.counter$ will be increased by 1.

```
GETKDOMITERMS(D, k)
1   if ( D = 1)
2     return 1
3   if ( D = 0 )
4     return NULL
5   else if ( D.term-list[k] exists )
6     return D.term-list[k]
7   else
8     COMPUTEKDOMINANTTERMS(D, k)
9   return D.term-list[k]
COMPUTEKDOMINANTTERMS(D, k)
01    if ( D = 1)
02      return 1
03    while (D.term-list[k] not exists) do
04      for each 0-edge linked vertex V starting with D do
05        if ( V = 1 and V.counter > 1)
06          continue
07        term = GETKDOMITERMS(V.child1, V.counter)
08        if ( term exists)
09          nterm = UPDATETERM(term, V)
10          tvalue = COMPUTETERMVALUE(nerm)
11          if (tvalue is the largest)
12            new_largest_term = nterm
13            vertex_need_update = V
14      if (new_largest_term exists)
15        D.term-list.push(new_largest_term)
16        vertex_need_update.counter++
17      else
18        break
19    return
```

Fig. 9. Dynamic programming based dominant term generation algorithm.

UPDATETERM() adds a vertex (its symbol) into a term represented by a DDD tree. COMPUTETERMVALUE() computes the numerical value of a given term. We use $V.child1$ to represent the vertex pointed to by the 1-edge originating from vertex V.

Let n be the number of vertices in a path from 1-terminal to the root vertex, i.e. the depth of the DDD graph. Let each circuit node be connected by at most m devices. Then each matrix entry can be a sum of at most m individual elements. The algorithm takes linear time in terms of the size of a DDD, if UPDATETERM() and COMPUTETERMVALUE() are implemented to use constant time each time when a vertex is added to a term. This can be accomplished by using memory caching. After obtaining the first dominant term, the next dominant can be found in $O(n)$ time assuming the UPDATETERM() and COMPUTETERMVALUE() take constant time and the number of 0-edge linked vertices is bounded by m and $m \ll n$ (due to line 04 in Fig. 9). But m depends on the circuit topologies and may become comparable to n for some strongly connected circuit structures and the time complexity of the DP algorithm will become $O(n^2)$ in this case.

We note that cancellation-free s-expanded DDDs do not satisfy Theorem 1. For example, Fig. 7 shows the cancellation-free s-expanded DDD of the s-expanded DDD in Fig. 6, vertex g in the coefficient of s^1 has both incoming 1-edge and incoming 0-edge. Verhaegen and Gielen [4], [23] resolved this problem by duplicating vertex g. Their approach, however, would destroy the DDD canonical property, a property that enables many efficient DDD-based graph manipulations. In this paper, we apply the proposed DP based term generation approach on the s-expanded DDDs before de-cancellation.

IV. CONSECUTIVE K-SHORTEST PATH ALGORITHM FOR GENERATION OF DOMINANT TERMS

We proposed an efficient algorithm for finding k dominant terms in [27]. The algorithm does not require DDDs to be *native* or to satisfy aforementioned graph theoretical property (Theorem 1), and thus can be applicable to any DDD graph. To differentiate this algorithm from the one presented in Section V, we refer to this algorithm as consecutive k-shortest path (SP) algorithm.

The SP algorithm is based on the observation that the most significant term in coefficient DDDs can be transformed into the shortest path in edge-weighted DDD graphs by introducing the following edge weight in a DDD:

- *0-edge* costs 0
- *1-edge* costs $-log|a_i|$, and $|a_i|$ denotes the numerical value of the DDD vertex a_i that originates the corresponding 1-edge.

The weight of a path in a coefficient DDD is defined to be the total weights of the edges along the path from the root to the 1-terminal. As a result, given a path, say $abcdef$, their path weight is

$$-(log|a| + log|b| + log|c| + log|d| + log|e| + log|f|). \quad (3)$$

If $|abcdef|$ is the value of the largest term, value of $-log|abcdef|$ will be the smallest, which actually is (3).

The shortest (weighted) path in a coefficient DDD, which is a DAG (direct acyclic graph), can be found by depth-first search in time $O(V + E)$, where V is the number of DDD vertices and E is number of edges [28]. So it is $O(V)$ in DDDs. Once we find the shortest path from a DDD, we can subtract it from the DDD using SUBTRACT() operation [21], and then we can find the next shortest path in the resulting DDD.

DDD function SUBTRACT() is the key operation in our SP algorithm. The pseudo code of SUBTRACT() operation is shown in Fig. 10.

Where, function GETVERTEX($top, child1.child0$) is to generate a vertex for a symbol top and two subgraphs $child1$ (pointed by 1-edge) and $child0$ (pointed by 0-edge) [21].

Let n be the number of vertices in a path from 1-terminal to the root vertex, i.e. the depth of the DDD graph, given the fact that D is a DDD graph and P is a path in the DDD form, then we have the following theorem:

Theorem 2: The number of new DDD vertices created in function SUBTRACT*(D, P) is bounded by n and the time complexity of the function is $O(n)$.*

Proof: As we know that DDD graph D contains path P. As P is a single-path DDD graph, $P.child0$ is always 0-terminal. So lines 5 and SUBTRACT($D.child0, P.child0$) in line 6 will immediately return D and $D.child0$ respectively (actually,

```
SUBTRACT(D, P)
1   if (D = 0) return 0
2   if (P = 0) return D
3   if (D = P) return 0
4   if (D.top > P.top)
        return GETVERTEX(D.top,
                P.child1,SUBTRACT(D.child0, P))
5   if (D.top < P.top) return SUBTRACT(D, P.child0)
6   if (D.top = P.top)
        return GETVERTEX(D.top,
                SUBTRACT(D.chinld1, P.child1),
                SUBTRACT(D.child0, P.child0))
```

Fig. 10. Implementation of SUBTRACT() for symbolic analysis and applications.

line 5 will never be reached if D contains path P). As a result, we will descend one level down in graph D each time depending on which line we choose to go for lines 4 and 6. In fact, function SUBTRACT(D, P) actually will traverse the *embedded* path P in D until it hits a common subgraph in both D and P as indicated in line 3. After this, $m-1$ new vertices will be created on its way back to the new root vertex, m is the number of vertices visited in D in the whole operation and $m < n$. If the common subgraph is 1-terminal, then $m = n$.

We then have following results:

Theorem 3: The time complexity of the SP algorithm for finding k shortest paths is

$$O(k|DDD| + n\frac{k(k-1))}{2}), \tag{4}$$

where n is the depth of the DDD graph.

Proof: According to Theorem 2, the number of new DDD vertices created in SUBTRACT() operation is bounded by n and the time complexity of the function is $O(n)$. Therefore, we can find the k shortest paths in time $O(k|DDD|+n(\sum_{i=1}^{k-1} i))$, which actually is Eq.(4).

V. INCREMENTAL K-SHORTEST PATH ALGORITHM FOR GENERATION OF DOMINANT TERMS

In this section, we introduce a more efficient term generation algorithm based on our previous work in [27]. The new algorithm is still based on the shortest path concept. But unlike the consecutive k-shortest path method, the new algorithm does not need to visit every vertex in a DDD graph to find the dominant term as required by the shortest path search algorithm [28] after every vertex has been visited once (i.e. after the first dominant term is found). The new algorithm is based on the observation that not all the vertices are needed to be visited, after the DDD graph is modified due to the subtraction of a dominant term from the graph. We show that only the newly added DDD vertices are needed to be relaxed and the number of newly added DDD vertices is bounded by the depth of a DDD graph.

In the sequel, we first introduce the concept of reverse DDD graphs. As shown in Fig. 4, a DDD graph is a direct graph with two terminal vertices and one root vertex. Remember that the 1-path in a DDD graph is defined from the root vertex to the 1-terminal. Now we define a new type of DDD graphs, called *reverse* DDD graphs where all the edges have their directions reversed and the root of the new graph are 1-terminal and 0-terminal vertices and new terminal vertex becomes the root vertex of the original DDD graph. The reverse DDD graph for the DDD graph in Fig. 4 is shown in Fig. 11. For the clarification, the root vertex and terminal vertices are still referred to as those in the original DDD graphs.

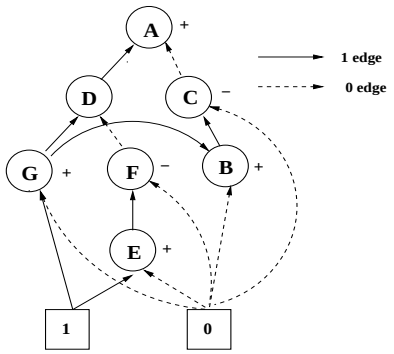

Fig. 11. A reverse DDD.

With the concept of the reverse DDD graph, we further define 1-path and path weight in a reverse DDD graph.

Definition 1: A *1-path* in a reverse DDD is defined as a path from the 1-terminal to root vertex (A in our example) including all symbolic symbols and signs of the vertices that the 1-edges point to along the 1-path.

Definition 2: The weight of a path in a DDD is defined to be the total weights of the edges along the path where each *0-edge* costs 0 and each *1-edge* costs $-log|a_i|$, and $|a_i|$ denotes the numerical value of the DDD vertex a_i that the corresponding 1-edge points to.

We then have the following result.

Lemma 2: The most significant product (dominant) term in a symbolic determinant D corresponds to the minimum cost (shortest) path in the corresponding reverse DDD between the 1-terminal and the root vertex.

The shortest path in a reverse s-expanded DDD, which is still a DAG and thus, can be found in $O(|DDD|)$ time as the normal DDD graph does.

Following the same strategy in [27], after we find the shortest path from a DDD, we can subtract it from the DDD using SUBTRACT() DDD operation, and then we can find the next shortest path in the resulting DDD. We have the following result:

Lemma 3: In a reverse DDD graph, after all the vertices have been visited (after finding the first shortest path), the next shortest path can be found by only visiting newly added vertices created by the subtraction operation.

Proof: The proof of Lemma 3 lies in the canonical nature of DDD graphs. A new DDD vertex is generated if and only if the subgraph rooted at the new vertex is a new and unique subgraph for the existing DDD graph. In other words, there do not exist two identical subgraphs in a DDD graph due to the canonical nature of DDD graphs. On the other hand, if an existing DDD vertex becomes part of the new DDD graph, its corresponding subgraphs will remain the same. As a result, the shortest path from 1-terminal to all vertices in the subgraph

will remain the same. Hence, it is sufficient to visit the newly added vertices to find the shortest paths from 1-terminal to those vertices. The root vertex of the new DDD graph is one of those newly added vertices.

Fig. 12 illustrates the incremental k-shortest path algorithm. The figure in the left-hand side shows consecutive k-shortest path algorithm to find the shortest path. Every time when a new DDD graph is created which is rooted at D', we have to visit the whole graph to find the shortest path. The figure shown in the right-hand side is the new incremental k-shortest path algorithm where we only need to visit all the newly created DDD nodes (in the upper left triangle) to be able to find the shortest path. As shortest paths are found from the source to all the nodes in a graph, the shortest paths, shown in dashed lines, in the existing subgraphs can be reused in the new DDD graph.

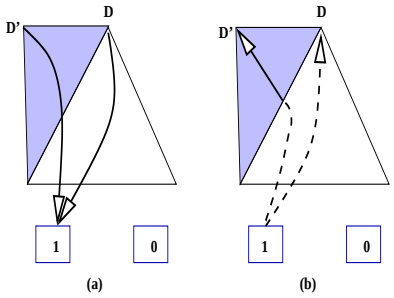

Fig. 12. Incremental k-shortest path algorithm.

It turns out that finding the shortest path from 1-terminal to the new vertices can be done very efficiently when those new vertices get created. The shortest path searching can virtually take no time during the subtraction operation. Suppose that every vertex in reverse DDD graph D has a shortest path from 1-terminal to it (be visited once). Then the new algorithm for searching the next dominant term is given in Fig. 13.

In GETNEXTSHORTESTPATH(D), EXTRACTPATH(D) obtains the found shortest path from D and returns the path in a single DDD graph form. This is done by simply traversing from the root vertex to 1-terminal. Each vertex will remember its immediate parent who is on the shortest path to the vertex in a fully relaxed graph (relaxation concept will be explained soon). Once the shortest path is found, we *subtract* it from the existing DDD graph and relax the newly created DDD vertices (line 15-17) at same time to find the shortest paths from 1-terminal to those vertices, which is performed in the modified function SUBTRACT(D, P), now called SUBTRACTANDRELAX(D, P).

In function SUBTRACTANDRELAX(D, P), RELAX(P, Q) performs the relaxation operation, an operation that checks if a path from a vertex's parent is the shortest path seen so far and remember the parent if it is, for vertices P and Q where P is the immediate parent of Q in the reverse DDD graph. The relaxation operation is shown in Fig. 14. Here, $d(x)$ is the shortest path value see so far for vertex x; $w(P, Q)$ is the weight of the edge from P to Q, which actually is the circuit parameter value that Q represents in the reverse DDD graph. Line $parent(Q) = P$ remembers the parent of Q in the

```
GETNEXTSHORTESTPATH(D)
1   if (D = 0)
2     return 0
3   P = EXTRACTPATH(D)
4   if (P exists and P not equal to 1)
5     D = SUBTRACTANDRELAX(D, P)
6   return P
SUBTRACTANDRELAXW(D, P)
01  if (D = 0)
02    return 0
03  if (P = 0)
04    return D
05  if (D = P)
06    return 0
07  if (D.top > P.top)
08    V = GETVERTEX(D.top, D.child1,
                    SUBTRACTANDRELAX(D.child0, P))
09  if (D.top < P.top)
10    V = SUBTRACTANDRELAX(D, P.child0)
11  if (D.top = P.top)
12    T1 = SUBTRACTANDRELAX(D.child1, P.child1))
13    T0 = SUBTRACTANDRELAX(D.child0, P.child0)
14    V = GETVERTEX(D.top, T1, T0)
15  if (V not equal to D)
16    RELAX(V.child1, V)
17    RELAX(V.child0, V)
18  return V
```

Fig. 13. Incremental k-shortest path based dominant term generation algorithm.

```
RELAX(P, Q)
1   if d(Q) > d(P) + w(P, Q)
2     d(Q) = d(P) + w(P, Q)
3     parent(Q) = P
```

Fig. 14. THE RELAX() operation.

shortest path from the 1-terminal to Q. In the reverse DDD graph, each vertex has only two incoming edges (from its two children in the normal DDD graph), so the relaxation with its two parents in lines 16 and 17 are sufficient for the relaxation of vertex V. Moreover, the relaxation for V happens after all its parents have been relaxed due to the DFS-type traversal in SUBTRACTANDRELAX(). This is consistent with the ordering requirement of the shortest path search algorithm. Therefore by repeatedly invoking function GETNEXTSHORTESTPATH(D), we can find all the dominant terms in a decreasing order. We have following result for incremental k-SP based algorithm:

Theorem 4: The time complexity of the incremental k-SP algorithm for finding k shortest paths is

$$O(|DDD| + n(k-1)), \tag{5}$$

where n is the depth of the DDD graph.

Proof: The RELAX() operation shown in Fig. 14 takes constant time to finish. As a result, the time complexity of the operation SUBTRACTANDRELAX() will still be $O(n)$ according to Theorem 2. After finding the first shortest path, which takes $O(|DDD|)$ to finish, the algorithm will take $O(n)$ to find each shortest path. Therefore, it takes $O(n(k-1))$ to find the rest of $k-1$ shortest paths and the total time complexity of finding the k shortest paths becomes $O(|DDD| + n(k-1))$.

Notice that both DP based algorithm and incremental k-SP

based algorithm have time complexity $O(|DDD|)$ to find a dominant term, where $|DDD|$ is the size of a DDD graph. After the first dominant term, however, both algorithms show better time complexities for generating next dominant terms, that is $O(n)$. But in contrast to DP based algorithm, the actual running time of the incremental k-SP based algorithm does not depend on the topology of a circuit.

Notice that the new incremental k-shortest path generation algorithm can be performed on any DDD graph, including cancellation-free s-expanded DDD. We note that the variant of DDD used by Verhaegen and Gielen in [4], [23] does not satisfy the canonical property due to vertex duplication. As a result, except for the first shortest path, remaining dominant paths cannot easily be generated by using the shortest path algorithm as the found shortest path is hard to be subtracted (if possible at all) as most DDD graph operations are not valid for a non-canonical DDD graph.

Following the same strategy in [5], our approach also handles numerical cancellation. Since numerical canceling terms are extracted one after another, they can be eliminated by examining two consecutive terms.

VI. Experimental Results

The proposed three algorithms have been implemented and tested on a number of practical analog circuits. For each circuit, DC analysis is first carried out using SPICE and our program reads in small-signal element values from the SPICE output. The algorithms described in [21], [22] are used to construct complex DDDs and s-expanded DDDs.

First we apply the proposed dominant term generation algorithms to derive interpretable symbolic expressions for transfer functions and poles from simple two-stage CMOS Opamp circuit shown in Fig. 1. The exact transfer function generated by our program is shown in Appendix . In the approximation process, we monitor both the magnitude and phase of the simplified expressions to control the accumulated error within a given frequency range. Before we generate dominant terms, we first simplify DDD graphs by device removal and node contraction on the complex DDD representation [24], [27], which will remove many insignificant terms and result in smaller complex DDDs.

For *TwoStage*, the simplified voltage gain for *TwoStage* given by our program is shown in Eq. (1). Table I shows the exact values of three zeros and three poles.

TABLE I
POLES AND ZEROS FOR OPAMP $TwoStage$.

poles	$-1.68 * 10^7$	$-8.80 * 10^5$	-373.74
zeros	$3.18 * 10^9$	$-1.68 * 10^7$	$1.11 * 10^7$

Since three poles are far away from each other, the pole splitting method can be used to find their symbolic expressions. For instance, the resulting expression of the first pole based on DDD manipulations is as follows:

$$-\frac{1}{(\frac{1}{r_{o2}}+\frac{1}{r_{o4}})(\frac{1}{r_{o6}}+\frac{1}{r_{o7}})g_{m6}C_C} = -373.74.$$

This agrees with the exact first pole described in Table I.

We then compare three dominant term generation algorithms – the consecutive k-shortest path based algorithm, dynamic shortest path based algorithm as well as the incremental k-shortest path based algorithm.

Table II summarizes the comparison results in terms of CPU time and memory usage for the three algorithms. A total of 10000 dominant terms are generated for a number of test circuits ranging from more regularly structured ladder circuits to less regularly structured such as $Cascode$ and $\mu A741$ amplifiers. In Table II, columns 1, 2 and 3 list for each circuit, respectively, its name *Circuit*, the number of nodes *#nodes*, and the number of nonzero elements *#nonzero* in its circuit MNA matrix. Columns 4 to 7 show, respectively, the CPU time and memory usage, for generating 10000 dominant terms by the DP based algorithm, *Dynamic Programming*, by the consecutive k-SP based algorithm, *k-Shortest Path*, respectively. The last column gives the CPU time of *Incr k-Shortest Path*. The memory usage for incremental k-shortest path is the same as consecutive k-SP algorithm for each circuit.

From Table II, we see that the incremental k-SP based algorithm consistently outperforms the DP based algorithm for all the circuits in both CPU time and memory usage. The difference becomes even more significant for circuits with regular structures like ladder circuits. Notice that for ladder circuits, $m \leq 3$. So $n \ll m$ is satisfied for large ladder circuits and the DP based algorithm is linear also after the first dominant term is found. For less regularly structured circuits like $\mu A741$, the incremental k-SP based method also shows impressive improvements over the DP based algorithm.

As we know that for both DP based algorithm and incremental k-SP based algorithm, the actual time to generate a new path is close to $O(n)$, where n is the size of the circuit or the depth of DDD graphs. But for the consecutive k-SP based algorithm, we have to visit all the vertices every time to generate a new path. Such difference is clearly demonstrated for circuits *Cascode* and $\mu A741$ where the sizes of DDDs are significantly larger than the sizes of the circuits. So the CPU time for the consecutive k-SP based algorithm is longer than that of two other algorithms.

Note that the CPU time of generating dominant terms by the shortest path depends on the sizes of s-expanded DDDs. As a result, ladder circuits should have shown a much better performance than less structured circuits like opamp circuits as DDD representations for ladder circuit are optimal. But as more terms are subtracted from the s-expanded DDDs, the sizes of resulting s-expanded DDDs will increase as a certain amount of sharing is destroyed. (According to Theorem 2, maximum n nodes may be added to the resulting DDD graphs after one SUBTRACTANDRELAX() operation, n is the size of the circuit or #nodes). Such increase of DDD sizes will become significant for ladder circuits as ladder circuits have the maximum sharing (optimal representations) at the beginning. Therefore, the CPU time for some large ladder circuits is comparable to that of some small opamp circuits. But for the incremental k-shortest path algorithm, after the first dominant term, the CPU time for generating a new dominant term is $O(n)$. So the CPU time is asymptotically proportional to the #nodes as shown in the Table II.

TABLE II

COMPARISON OF SHORTEST-PATH AND DYNAMIC-PROGRAMMING BASED ALGORITHMS.

Circuit	#nodes	#nonzero	Dynamic Programming		k-Shortest Path		Incr k-Shortest Path
			CPU time (sec.)	Memory use (kb)	CPU time (sec.)	Memory Use (kb)	
rclad10	8	31	17.3	9560	14.8	4048	5.1
rclad21	22	64	105.1	21816	21.5	3976	15.4
rclad60	61	181	133.8	51784	101.0	5432	80.3
rclad100	101	301	369.6	73832	172.8	10568	132.9
rclad150	151	451	912.2	253072	281.7	23280	248.3
rclad200	201	601	1630.9	365264	387.3	39616	320.0
rclad250	251	751	2431.3	481688	493.8	60160	426.4
rclad300	301	901	3388.4	602360	598.0	85088	557.4
rctreeA	40	119	41.4	38840	45.6	12880	41.0
rctreeB	53	158	132.9	41472	60.4	14304	57.2
Cascode	14	76	21.3	34880	620.1	28696	15.3
$\mu A741$	23	90	50.6	78024	1412.2	69184	21.0
bigtst	32	112	91.7	40808	144.1	12496	32.7

Fig. 15 shows the CPU time for different ladder circuits. The CPU time increases almost linearly with the size of a ladder circuit for all three algorithms. Both the SP-based algorithms consistently outperforms DP based algorithm in terms of CPU time. The reason is that sizes of DDDs for representing ladder circuits grow linearly with the sizes of the ladder circuits, that is n [21], so the time complexities of all three algorithms, $O(|DDD|)$, become $O(n)$. But the DP based algorithm need to take extra efforts to loop through all 0-linked vertices to compute the dominant terms and restore them at each 1-edged pointed vertex. Those extra efforts will become significantly when the graph become very deep as with the higher order ladder circuits.

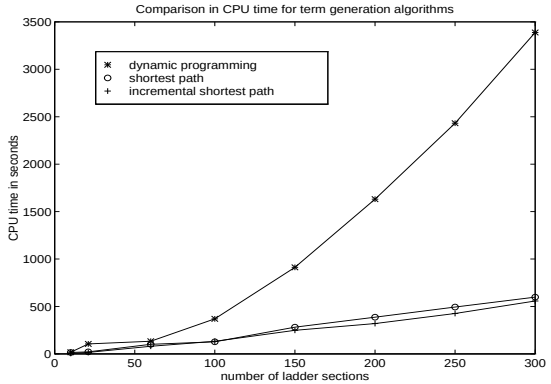

Fig. 15. CPU time vs number of ladder sections.

Fig. 16 shows the memory usage for different ladder circuits. The memory usage of DP based method increases linearly with the sizes of ladder circuits, while the consecutive k-SP and incremental k-SP algorithms take much smaller amount of memory for the same set of ladder circuits. The DP based method will use more memory when more terms are generated as more memory will be used for caching the generated terms at each 1-edge pointed DDD vertex.

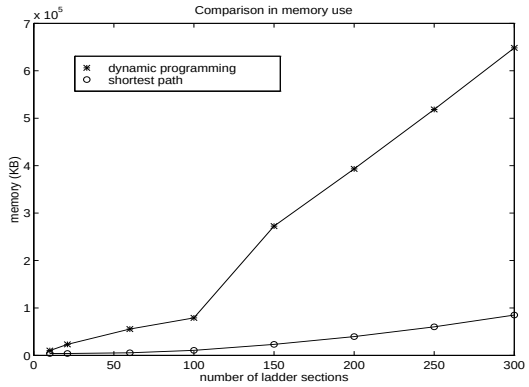

Fig. 16. Memory use vs number of ladder sections.

VII. CONCLUSIONS

Efficient algorithms were proposed to generate dominant terms for ac characteristics of large linear analog circuits. The new algorithms are based on a DDD graph-based compact and canonical representation of symbolic expressions. We formulate the dominant term generation problem as the one of searching for the k shortest paths in DDD graphs. Theoretically we showed that dynamic programming based dominant term generation method is restricted to certain DDD graphs. Practically, we proposed a incremental k-shortest search algorithm, which can be applied to any DDD graphs, based on the canonical property of DDD graphs. Experimental results indicate that the proposed incremental k-shortest path algorithm outperform the best known algorithm based on the dynamic programming [4], [23] in terms of CPU time and memory usage.

APPENDIX

Exact transfer function of two-stage CMOS opamp in Fig. 1 The numerical value at the end of each line is the magnitude of the coefficient of $s^k, k = 1...n$ if the line starts with s^k or the magnitude of the product term in the same line otherwise. All the products in each coefficient are sorted with respect to their numerical magnitudes.

```
numerator:
(-) * [
s^0(-2.128e-14
+ (gm2) (gm3) (gm6) ,-2.1e-14
+ (gm2) (1/ro1+ro3) (gm6) ,-2.8e-16
) +
s^1(1.03464e-22
+ (gm2) (gm3) (-cc) ,3e-22
+ (gm2) (cgd1+cdb1+cgs3+cdb3+cgs4) (gm6) ,-2.002e-22
+ (gm2) (1/ro1+ro3) (-cc) ,4e-24
+ (gm2) (cgd4) (gm6) ,-1.4e-24
+ (-cgd2) (gm3) (gm6) ,1.05e-24
+ (-cgd2) (1/ro1+ro3) (gm6) ,1.4e-26
) +
s^2(2.87488e-30
+ (gm2) (cgd1+cdb1+cgs3+cdb3+cgs4) (-cc) ,2.86e-30
+ (gm2) (cgd4) (-cc) ,2e-32
+ (-cgd2) (gm3) (-cc) ,-1.5e-32
+ (-cgd2) (cgd1+cdb1+cgs3+cdb3+cgs4) (gm6) ,1.001e-32
+ (-cgd2) (1/ro1+ro3) (-cc) ,-2e-34
+ (-cgd2) (cgd4) (gm6) ,7e-35
) +
s^3(-1.44e-40
+ (-cgd2) (cgd1+cdb1+cgs3+cdb3+cgs4) (-cc) ,-1.43e-40
+ (-cgd2) (cgd4) (-cc) ,-1e-42
) +
number of terms:16
]
----------------------------
denominator:
(+) * [
s^0(2.61578e-18
+ (gm3) (1/ro2+ro4) (1/ro6+ro7) ,2.58136e-18
+ (1/ro1+ro3) (1/ro2+ro4) (1/ro6+ro7) ,3.44182e-20
) +
s^1(1.21479e-21
- (gm3) (gm6) (-cc) ,1.05e-21
+ (gm3) (cdb2+cdb4+cgs6+cl) (1/ro6+ro7) ,1.32837e-22
- (1/ro1+ro3) (gm6) (-cc) ,1.4e-23
+ (gm3) (cc) (1/ro6+ro7) ,1.29068e-23
+ (gm3) (1/ro2+ro4) (cc) ,3e-24
+ (1/ro1+ro3) (cdb2+cdb4+cgs6+cl) (1/ro6+ro7) ,1.77116e-24
+ (1/ro1+ro3) (cc) (1/ro6+ro7) ,1.72091e-25
+ (1/ro1+ro3) (1/ro2+ro4) (cc) ,4e-26
+ (cgd1+cdb1+cgs3+cdb3+cgs4) (1/ro2+ro4) (1/ro6+ro7) ,2.4609e-26
- (gm4) (-cgd4) (1/ro6+ro7) ,1.29068e-26
+ (gm3) (cgd4) (1/ro6+ro7) ,1.29068e-26
+ (gm3) (cgd2) (1/ro6+ro7) ,1.29068e-26
+ (gm3) (1/ro2+ro4) (cdb6) ,3e-27
+ (1/ro1+ro3) (cgd4) (1/ro6+ro7) ,1.72091e-28
+ (1/ro1+ro3) (cgd2) (1/ro6+ro7) ,1.72091e-28
+ (cgd4) (1/ro2+ro4) (1/ro6+ro7) ,1.72091e-28
+ (1/ro1+ro3) (1/ro2+ro4) (cdb6) ,4e-29
) +
s^2(1.68164e-28
+ (gm3) (cdb2+cdb4+cgs6+cl) (cc) ,1.5438e-28
- (cgd1+cdb1+cgs3+cdb3+cgs4) (gm6) (-cc) ,1.001e-29
+ (1/ro1+ro3) (cdb2+cdb4+cgs6+cl) (cc) ,2.0584e-30
+ (cgd1+cdb1+cgs3+cdb3+cgs4) (cdb2+cdb4+cgs6+cl) (1/ro6+ro7) ,1.26638e-30
+ (gm3) (cdb2+cdb4+cgs6+cl) (cdb6) ,1.5438e-31
+ (cgd1+cdb1+cgs3+cdb3+cgs4) (cc) (1/ro6+ro7) ,1.23045e-31
- (cgd4) (gm6) (-cc) ,7e-32
+ (cgd1+cdb1+cgs3+cdb3+cgs4) (1/ro2+ro4) (cc) ,2.86e-32
+ (gm4) (-cgd4) (cc) ,1.5e-32
+ (gm3) (cgd4) (cc) ,1.5e-32
+ (gm3) (cgd2) (cc) ,1.5e-32
+ (gm3) (cc) (cdb6) ,1.5e-32
+ (cgd4) (cdb2+cdb4+cgs6+cl) (1/ro6+ro7) ,8.8558e-33
+ (1/ro1+ro3) (cdb2+cdb4+cgs6+cl) (cdb6) ,2.0584e-33
+ (cgd4) (cc) (1/ro6+ro7) ,8.60455e-34
+ (1/ro1+ro3) (cgd4) (cc) ,2e-34
+ (1/ro1+ro3) (cgd2) (cc) ,2e-34
+ (cgd4) (1/ro2+ro4) (cc) ,2e-34
+ (1/ro1+ro3) (cc) (cdb6) ,2e-34
+ (cgd1+cdb1+cgs3+cdb3+cgs4) (cgd4) (1/ro6+ro7) ,1.23045e-34
+ (cgd1+cdb1+cgs3+cdb3+cgs4) (cgd2) (1/ro6+ro7) ,1.23045e-34
+ (cgd1+cdb1+cgs3+cdb3+cgs4) (1/ro2+ro4) (cdb6) ,2.86e-35
+ (gm4) (-cgd4) (cdb6) ,1.5e-35
+ (gm3) (cgd4) (cdb6) ,1.5e-35
+ (gm3) (cgd2) (cdb6) ,1.5e-35
+ (cgd4) (cgd2) (1/ro6+ro7) ,8.60455e-37
+ (1/ro1+ro3) (cgd4) (cdb6) ,2e-37
+ (1/ro1+ro3) (cgd2) (cdb6) ,2e-37
+ (cgd4) (1/ro2+ro4) (cdb6) ,2e-37
) +
s^3(1.48396e-36
+ (cgd1+cdb1+cgs3+cdb3+cgs4) (cdb2+cdb4+cgs6+cl) (cc) ,1.47176e-36
+ (cgd4) (cdb2+cdb4+cgs6+cl) (cc) ,1.0292e-38
+ (cgd1+cdb1+cgs3+cdb3+cgs4) (cdb2+cdb4+cgs6+cl) (cdb6) ,1.47176e-39
+ (cgd1+cdb1+cgs3+cdb3+cgs4) (cc) (cdb6) ,1.43e-40
+ (cgd1+cdb1+cgs3+cdb3+cgs4) (cgd4) (cc) ,1.43e-40
+ (cgd1+cdb1+cgs3+cdb3+cgs4) (cgd2) (cc) ,1.43e-40
+ (cgd4) (cdb2+cdb4+cgs6+cl) (cdb6) ,1.0292e-41
+ (cgd4) (cc) (cdb6) ,1e-42
+ (cgd4) (cgd2) (cc) ,1e-42
+ (cgd1+cdb1+cgs3+cdb3+cgs4) (cgd4) (cdb6) ,1.43e-43
+ (cgd1+cdb1+cgs3+cdb3+cgs4) (cgd2) (cdb6) ,1.43e-43
+ (cgd4) (cgd2) (cdb6) ,1e-45
) +
number of terms:60
]
```

ACKNOWLEDGMENT

The authors would like to thanks four anonymous reviewers for their careful reviews of this paper and for their constructive comments.

REFERENCES

[1] W. Daems, W. Verhaegen, P. Wambacq, G. Gielen, and W. Sansen, "Evaluation of error-control strategies for the linear symbolic analysis of analog integrated circuits," *IEEE Trans. on Circuits and Systems I: Fundamental Theory and Applications*, vol. 46, no. 5, pp. 594–606, May 1999.

[2] F. V. Fernández, P. Wambacq, G. Gielen, A. Rodríguez-Vázquez, and W. Sansen, "Symbolic analysis of large analog integrated circuits by approximation during expression generation," in *Proc. IEEE Int. Symp. on Circuits and Systems (ISCAS)*, 1994, pp. 25–28.

[3] F. Leyn, G. Gielen, and W. Sansen, "Analog small-signal modeling - part I: behavioral signal path modeling for analog integrated circuits," *IEEE Trans. on Circuits and Systems II: analog and digital signal processing*, vol. 48, no. 7, pp. 701–711, July 2001.

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

[5] P. Wambacq, G. Gielen, and W. Sansen, "A cancellation-free algorithm for the symbolic simulation of large analog circuits," in *Proc. IEEE Int. Symp. on Circuits and Systems (ISCAS)*, May 1992, pp. 1157–1160.

[6] ——, "A new reliable approximation method for expanded symbolic network functions," in *Proc. IEEE Int. Symp. on Circuits and Systems (ISCAS)*, 1996, pp. 584–587.

[7] Q. Yu and C. Sechen, "A unified approach to the approximate symbolic analysis of large analog integrated circuits," *IEEE Trans. on Circuits and Systems I: Fundamental Theory and Applications*, vol. 43, no. 8, pp. 656–669, Aug. 1996.

[8] H. Floberg, *Symbolic Analysis in Analog Integrated Circuit Design*. Massachusetts: Kluwer Academic Publisher, 1997.

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

[10] G. Gielen, P. Wambacq, and W. Sansen, "Symbolic analysis methods and applications for analog circuits: A tutorial overview," *Proc. of IEEE*, vol. 82, no. 2, pp. 287–304, Feb. 1994.

[11] W. Verhaegen and G. Gielen, "Symbolic distortion analysis of analog integrated circuits," in *Proc. European Conference on Circuit Theory and Design*, Aug. 2001, pp. I.21–I.24.

[12] P. Wambacq and W. Sansen, *Distortion Analysis of Analog Integrated Circuits*. Kluwer Academic Publishers, 1998.

[13] P. Vanassche, G. Gielen, and W. Sansen, "Symbolic modeling of periodically time-varying systems using harmonic transfer functions," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 21, no. 9, pp. 1011–1024, Sept. 2002.

[14] P. M. Lin, *Symbolic Network Analysis*. Elsevier Science Publishers B.V., 1991.

[15] S. J. Seda, M. G. R. Degrauwe, and W. Fichtner, "A symbolic analysis tool for analog circuit design automation," in *Proc. Int. Conf. on Computer Aided Design (ICCAD)*, Nov. 1988, pp. 488–491.

[16] F. V. Fernández, J. Martín, A. Rodríguez-Vázquez, and J. L. Huertas, "On simplification techniques for symbolic analysis of analog integrated circuits," in *Proc. IEEE Int. Symp. on Circuits and Systems (ISCAS)*, 1992, pp. 1149–1152.

[17] S. J. Seda, M. G. R. Degrauwe, and W. Fichtner, "Lazy-expansion symbolic expression approximation in synap," in *Proc. Int. Conf. on Computer Aided Design (ICCAD)*, Nov. 1992, pp. 310–317.

[18] J.-J. Hsu and C. Sechen, "Dc small signal symbolic analysis of large analog integrated circuits," *IEEE Trans. on Circuits and Systems I: Fundamental Theory and Applications*, vol. 41, no. 12, pp. 817–828, Dec. 1994.

[19] W. Daems, G. Gielen, and W. Sansen, "Circuit simplification for the symbolic analysis of analog integrated circuits," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 21, no. 4, pp. 395–407, April 2002.

[20] F. Leyn, G. Gielen, and W. Sansen, "Analog small-signal modeling - part II: elementary transistor stages analyzed with behavioral signal path modeling," *IEEE Trans. on Circuits and Systems II: analog and digital signal processing*, vol. 48, no. 7, pp. 701–711, July 2001.

[21] 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, no. 1, pp. 1–18, Jan. 2000.

[22] ——, "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, no. 7, pp. 813–827, April 2001.

[23] W. Verhaegen and G. Gielen, "Symbolic determinant decision diagrams and their use for symbolic modeling of linear analog integrated circuits," *Kluwer International Journal on Analog Integrated Circuits and Signal Processing*, vol. 31, no. 2, pp. 119–130, May 2002.

[24] S. X.-D. Tan and C.-J. Shi, "Parametric analog behavioral modeling based on cancellation-free ddds," in *Proc. IEEE International Workshop on Behavioral Modeling and Simulation (BMAS)*, Oct. 2002.
[25] ——, "Efficient ddd-based term generation algorithm for analog circuit behavioral modeling," in *Proc. Asia South Pacific Design Automation Conf. (ASPDAC)*, Jan. 2003, pp. 789–794.
[26] C.-J. Shi and X.-D. Tan, "Efficient derivation of exact s-expanded symbolic expressions for behavioral modeling of analog circuits," in *Proc. IEEE Custom Integrated Circuits Conference (CICC)*, 1998, pp. 463–466.
[27] X.-D. Tan and C.-J. Shi, "Interpretable symbolic small-signal characterization of large analog circuits using determinant decision diagrams," in *Proc. European Design and Test Conf. (DATE)*, 1999, pp. 448–453.
[28] T. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. Cambridge, Massachusetts: The MIT Press, 1990.

Sheldon X.-D. Tan (S'96-M'99) received his 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 was a faculty member in the electrical engineering Department of Fudan University from 1995 to 1996. He worked for Monterey Design Systems Inc. CA, from 1999 to 2001 and Altera Corporation CA, from 2001 to 2002. Now he is with the Department of Electrical Engineering, University California at Riverside as an Assistant Professor.

His research interests include several aspects of design automation for VLSI integrated circuits – modeling, analysis and optimization of mixed-signal/RF/analog circuits, high-performance and intelligent embeded systems, signal integrity issues in VLSI physical design, high performance power/ground distribution network design and optimization.

Dr. Tan received a Best Paper Award from 1999 IEEE/ACM Design Automation Conference. He also received Best Graduate Award and a number of Outstanding College Student Scholarships from Fudan University.

C.-J. Richard Shi (M'91-SM'99) is an Associate Professor in Electrical Engineering at the University of Washington. His research interests include several aspects of the computer-aided design and test of integrated circuits and systems, with particular emphasis on analog/mixed-signal and deep-submicron circuit modeling, simulation and design automation. Dr. Shi is a key contributor to IEEE std 1076.1-1999 (VHDL-AMS) standard for the description and simulation of mixed-signal circuits and systems. He founded IEEE International Workshop on Behavioral Modeling and Simulation (BMAS) in 1997, and has served on the technical program committees of several international conferences.

Dr. Shi received a Best Paper Award from the IEEE/ACM Design Automation Conference, a Best Paper Award from the IEEE VLSI Test Symposium, a National Science Foundation CAREER Award, and a Doctoral Prize from the Natural Science and Engineering Research Council of Canada. He has been an Associate Editor, as well as a Guest Editor, of the IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS-II, ANALOG AND DIGITAL SIGNAL PROCESSING. He is currently an Associate Editor of IEEE Transactions on Computer-Aided Design of Integrate Circuits and Systems.