Submitted to IEEE Custom Integrated Circuits Conference, (CICC'98), Santa Clara, CA, May 11-14, 1998

Efficient Derivation of Exact s-Expanded Symbolic Expressions for Behavioral Modeling of Analog Circuits*

C.-J. Richard Shi and Xiangdong Tan
Department of Electrical and Computer Engineering
University of Iowa, Iowa City, Iowa 52242, U.S.A.

Abstract—**This paper presents an efficient method for deriving exact s-expanded symbolic expressions for behavioral modeling of analog circuits. The key idea is to introduce a graph-based representation called multi-root Determinant Decision Diagram (DDD), where each root represents the symbolic coefficient of a particular power of s. By exploiting sharing and sparsity, a huge number of symbolic product terms can be compactly represented and efficiently manipulated. Experimental results have demonstrated that this method can produce the exact s-expanded-symbolic transfer function for μA741 opamps in several CPU seconds on a UltraSparc-I workstation, and the use of generated symbolic expression for frequency-domain simulation achieved a speedup of about 50 over SPICE3F5.**

I. Introduction

Symbolic analysis is to calculate the behavior or the characteristic of a circuit in terms of symbolic parameters. Due to its importance in many applications such as optimum topology selection, design space exploration, behavioral model generation, and fault detection, considerable amount of research has been conducted in the 1960s-1980s [5]. Recently, there has been a great surge of interest in symbolic analysis [1, 2, 3].

Among several forms of symbolic representation, the s-expanded form is most useful. For example, a circuit transfer function $H(s)$ can be written in the following s-expanded form:

$$H(s) = \frac{\sum f_i(p_1, p_2, \cdots, p_m)s^i}{\sum g_j(p_1, p_2, \cdots, p_m)s^j} \qquad (1)$$

where $f_i(p_1, p_2, \cdots, p_m)$ and $g_j(p_1, p_2, \cdots, p_m)$ are symbolic functions in terms of symbolic circuit parameters $p_1, p_2, \cdots, p_m$ and do not contain the complex variable s. With s-expanded representations, symbolic poles and zeros can be approximated using the pole-splitting method [2]. With some or all circuit parameters substituted with numerical values, frequency response calculation using the s-expanded representation can be much faster than SPICE-like methods [5].

Despite of the usefulness of the s-expanded symbolic representation, no efficient derivation method exists. The interpolation method usually requires all the circuit parameters to be numerical and s is the only symbolic variable. Topological methods, such as the signal-flow graph method and tree enumeration method, can generate the completely expanded representation in the form of sum of product of symbolic parameters. However, the number of expanded product terms grows exponentially with the size of the circuit. Various simplification schemes can be used to find approximate symbolic expressions [1, 3], however, it is well known that approximate expressions may not be adequate for complete circuit characterization such as symbolic pole-zero derivation [3]. Arbitrarily nested hierarchical expressions or sequences of expressions can be useful for circuit simulation [4], but no method exists to derive the s-expanded expressions.

In this paper, we describe a new approach to the generation of the exact s-expanded symbolic expressions for behavioral modeling of analog circuits. The basic idea is to exploit a recently proposed graph-based representation for symbolic determinants called Determinant Decision Diagram (DDD) [6].

After reviewing the concept of DDDs in Section II, we describe in Section III how all the symbolic coefficients of the power of s can be elegantly represented by a multi-root DDD. An efficient procedure is presented in Section IV for deriving the DDD for the original complex DDD. Section V reports experimental results.

II. Determinant Decision Diagrams

Formally, a determinant decision diagram is a signed rooted directed acyclic graph with two terminal ver-

tices, namely the 0-terminal vertex and the 1-terminal vertex. Each vertex, labeled as a matrix entry a_i, represents a matrix determinant D, and it has two outgoing edges, called 1-edge and 0-edge, pointing, respectively, to the two vertices representing determinant D_{a_i} and $D_{\overline{a}_i}$. A determinant graph having root vertex a_i denotes a matrix determinant D defined recursively as follows:

1. if a_i is the 1-terminal vertex, then $D = 1$,
2. if a_i is the 0-terminal vertex, then $D = 0$,
3. if a_i is a non-terminal vertex, then $D = a_i\ s(a_i)\ D_{a_i} + D_{\overline{a}_i}$.

Here $s(a_i)$ is a sign $\{+, -\}$ associated with each non-terminal vertex a_i. It can be computed recursively as follows.

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

$$s(v) = \prod_{x \in P(v)} sign(r(x) - r(v))\ sign(c(x) - c(v)), \tag{2}$$

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

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

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

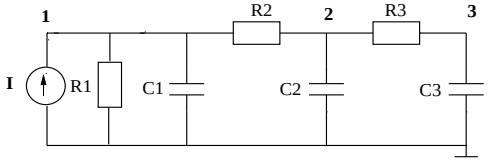

Figure 1: An example circuit.

For example, consider a simple circuit shown in Fig. 1. Its system equations can be written as

$$\begin{bmatrix} \frac{1}{R_1} + {}_sC_1 + \frac{1}{R_2} & -\frac{1}{R_2} & 0 \\ -\frac{1}{R_2} & \frac{1}{R_2} + {}_sC_2 + \frac{1}{R_3} & -\frac{1}{R_3} \\ 0 & -\frac{1}{R_3} & \frac{1}{R_3} + {}_sC_3 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} I \\ 0 \\ 0 \end{bmatrix}$$

To simply the discussion, we rewrite the circuit matrix as follow:

$$\begin{bmatrix} a + bs & c & 0 \\ d & e + fs & g \\ 0 & h & i + js \end{bmatrix}$$

Figure 2 shows a DDD with matrix entries as vertices. Each vertex is also labeled by a sign determined by the sign rule.

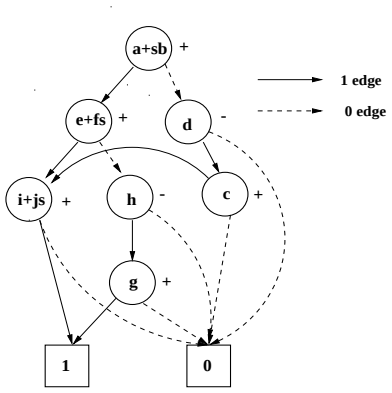

Figure 2: Complex DDD for a matrix determinant.

We define the 1-path as the set of 1-edges in a path from the root to the 1-terminal. Then it can be seen that each 1-path represents a symbolic product term. In this example, there are three product terms:

$$(a + sb)(e + fs)(i + js)$$
$$(a + sb)(-h)(g)$$
$$(-d)(c)(i + js)$$

The root of this DDD represents the sum of these three product terms, which is the expanded form of the symbolic determinant. Since complex variable s appears in some vertices, we refer to this DDD as the *complex* DDD.

III. s-Expanded Symbolic Representation

We consider how to expand a symbolic expression in s. For this example, we have

$$(a + sb)(e + fs)(i + js) \rightarrow \begin{cases} +aeis^0 \\ +aejs^1 \\ +afis^1 \\ +beis^1 \\ +afjs^2 \\ +bejs^2 \\ +bfis^2 \\ +bfjs^3 \end{cases}$$

$$(a + sb)(-h)(g) \rightarrow \begin{cases} -ahgs^0 \\ -bhgs^1 \end{cases}$$

$$(-d)(c)(i + js) \rightarrow \begin{cases} -dcis^0 \\ -dcjs^1 \end{cases}$$

Unfortunately, the derivation and representation of such expressions become quickly unmanageably with the increase of the circuit size. Our idea to resolve this problem is to introduce a multi-root DDD to represent all the coefficients of polynomials in s. Figure 3 is a DDD with four roots, representing the coefficient symbolic expressions for s^0, s^1, s^2, and s^4. The 0-terminal is omitted. It can be verified that this multi-root DDD represents exactly all the s-expanded symbolic product terms (12 terms). We refer to this multi-root DDD as the coefficient DDD.

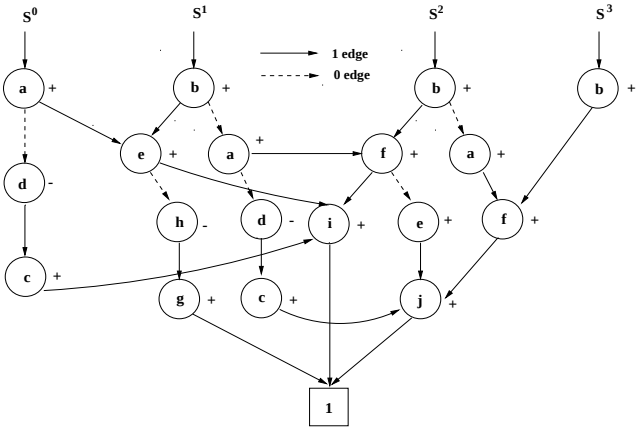

Figure 3: Coefficient DDD for a matrix determinant.

This representation enables the sharing among different coefficients in a polynomial. Further, the sharing among both the denominator and numerator of the same transfer function, and sharing across different transfer functions and analyses, can be exploited. In Fig. 3, 18 non-terminal vertices are used. In comparison, without exploiting sharing and sparsity, 12*9 (#product-terms $\times$ #symbols) vertices would be used.

IV. Construction of s-expanded DDD

Now, we describe formally a recursive procedure for creating the s-expanded coefficient DDD from a complex DDD. The procedure is very efficient, and has the time complexity proportional to the size of the complex DDD, not the number of product terms.

Let D be a complex DDD vertex, with its 1-edge pointing to D_1 and its 0-edge pointing to D_0. Each complex-DDD vertex corresponds to an entry in the original MNA matrix, and let it be in the form of $D.g + sD.c + \frac{1}{sD.l}$, where $D.g$, $D.c$ and $D.l$ are real and denote, respectively, the contributions from the conductance, capacitance and inductance components in the circuit. We introduce the following four basic operations:

- CoeffUnion(P_1, P_2) computes the union of two coefficient DDDs, P_1 and P_2.
- CoeffMulplty(P, v) computes the product of coefficient DDD P and coefficient DDD vertex v.
- $P * s$ increments the power of s in coefficient DDD P.
- P/s decrements the powers of s in coefficient DDD P.

Algorithm CoeffConstruction described in Fig. 4 takes a complex DDD vertex and creates its corresponding coefficient DDD. Similar to all other DDD operations [6], we cache the result of CoeffConstruction(D), and in case D is re-visited, and its result will be used directly.

```
CoeffConstruction(D)
1   if ( D = 0 or D = 1)
2       return NULL
3   L0 = CoeffConstruction(D0)
4   L1 = CoeffConstruction(D1)
5   if (D.g ≠ 0)
6       Pg = CoeffMulplty(L1, D.g)
7   if (D.c ≠ 0)
8       Pc = CoeffMulplty(L1 * s, D.c)
9       Presult = CoeffUnion(Pc, Pg)
10  if (D.l ≠ 0)
11      Pl = CoeffMulplty(L1/s, D.l)
12      Presult = CoeffUnion(Pl, Presult)
13  return CoeffUnion(Presult, L0)
```

Figure 4: Coefficient DDD construction.

V. Results

We have implemented the proposed method and tested on a set of benchmark circuits (including MOS circuits, bipolar, and BiCMOS circuits). For each circuit, the MNA formulation is used, and the algorithm described in [6] is applied to construct the complex DDD of the MNA matrix. The results are summarized in Table 1.

From Table 1, we can see that DDD is capable of representing a huge number of product terms with a relatively small number of vertices. The difference is even more dramatic for coefficient DDDs, where the

Table 1: Statistics on complex DDD and coefficient DDD construction.

circuit	#n	#e	Complex DDD				Coeff DDD			
			#numP	#denP	$\|DDD\|$	CPU	#numP	#denP	$\|DDD\|$	CPU
miller	7	23	9	13	53	0.02	92	252	461	< 0.01
butter	8	19	1	13	24	0.01	1	239	122	< 0.01
rctest	10	37	80	224	137	0.04	140	444	220	0.01
vcstest	10	37	52	132	85	0.04	80	372	122	0.02
cascode	15	77	42154	79643	2058	3.17	9.28×10^6	1.52×10^7	21384	0.88
$\mu A741$	24	89	10970	108032	6656	2.96	7.77×10^{10}	9.31×10^{10}	99846	5.08
rcnet	33	114	9.66×10^5	1.66×10^7	2844	12.01	1.49×10^{10}	8.98×10^{11}	31454	1.51

For each **circuit**, **#n** is the number of nodes in the circuit, and **#e** the number of non-zero elements in the MNA matrix.
#numP is the total number of product terms, i.e., 1-paths, in the numerator of the transfer function.
#demP is the total numbers of product terms, i.e., 1-paths, in the denominator of the transfer function.
|**DDD**| is the size (number of vertices) of the DDD representing both the numerator and denominator of the transfer function.
The **CPU** time is given in seconds on an UltraSparc-I workstation.

sharing among different coefficients of polynomials for both the numerator and denominator of the transfer function are exploited. Figure 5 shows the distribution of the number of product terms over power s in the denominator of the transfer function of the μA741 opamp. The construction of DDDs takes only a few CPU seconds on a UltraSparc-I workstation for all the examples.

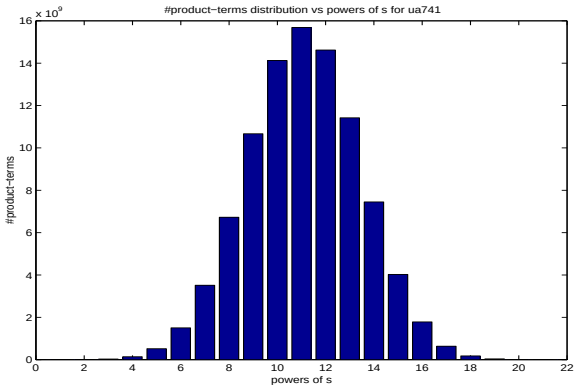

Figure 5: Product term distribution of μA741.

We implemented a frequency-domain simulator based on the derived s-expanded expressions. Table 2 describes the CPU time used, where *Coeff_Eval* is the time used for calculating the numerical values of a coefficient DDD, *Coeff DDD* is the CPU time used for frequency-domain simulation using the calculated s-expanded expressions. We observe that an order-of-magnitude speedup has been achieved, and further the speedup increases with the size of the circuit.

Table 2: Comparison of the new method with Spice.

circuit	Coeff_Eval	Coeff DDD	Spice	speedup
miller	< 0.01	0.02	0.32	16.0
butter	< 0.01	0.03	0.54	18.0
rctest	0.01	0.03	0.44	14.7
vcstest	< 0.01	0.02	0.52	26.0
cascode	0.52	0.03	1.16	38.6
$\mu A741$	3.32	0.05	2.40	48.0
rcnet	1.72	0.07	3.76	53.7

References

[1] F. V. Fernández and A. Rodríguez-Vázquez "Symbolic analysis tools—the state of the art", pp. 798–801 in *Proc. IEEE Int. Symposium on Circuits and System*, 1996.

[2] H. Floberg, *Symbolic Analysis in Analog Integrated Circuit Design*, Kluwer Academic Publishers, 1997.

[3] 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.

[4] M. M. Hassoun and P. M. Lin, "A hierarchical network approach to symbolic analysis of large scale networks", *IEEE Trans. Circuits and Systems*, vol. 42, no. 4, pp. 201–211, April 1995.

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

[6] C. J. Richard Shi, X. Tan, "Symbolic analysis of large analog circuits with determinant decision diagrams", pp. 366-373 in *Proc. IEEE Int. Conf. Computer Aided Design (ICCAD)*, 1997.