Incremental Symbolic Construction for Topological Modeling of Analog Circuits

Hanbin Hu[1], Guoyong Shi[2] and Yan Zhu[3]
School of Microelectronics, Shanghai Jiao Tong University, Shanghai 200240, China
Email: [1]huhanbinnew@hotmail.com, [2]shiguoyong@ic.sjtu.edu.cn, [3]newzhuyan@gmail.com

***Abstract*—Symbolic methods for analog circuit analysis and modeling have been well studied. However, little is known on how to create symbolic models incrementally while a circuit topology is being modified. This paper proposes an incremental symbolic construction method applicable to incremental circuit topology change based on a previously developed data structure called GPDD (graph-pair decision diagram). An incremental GPDD algorithm (*iGPDD*) is proposed. It is demonstrated experimentally that with proper symbol ordering the *iGPDD* method outperforms the restarted GPDD construction method.**

***Index Terms*—Binary decision diagram (BDD), graph-pair decision diagram (GPDD), incremental construction, infinite symbol, symbol ordering.**

I. Introduction

Over half a century symbolic circuit analysis has been studied mainly for behavioral modeling and design insights [1]–[3]. The introduction of binary decision diagram (BDD) to symbolic analysis of analog circuits has made it possible to build faster and more compact symbolic representations [4]–[6].

Analog integrated circuit design must deal with a variety of design goals simultaneously. It is sometimes necessary to change the circuit topology for realizing certain design goals. Traditionally, the designers would start from choosing a topology template, then modifying the template by adding or subtracting certain substructures in hope of achieving the design goals. It is highly desirable to develop a computerized tool to automate such topological exploration. However, such efforts are rarely reported in the literature except for the recent work [7] which proposed a symbolic signal-flow method for systematically comparing circuit topologies to identify the performance differences.

This paper presents an incremental symbolic construction method for analog circuits by the means of Graph-Pair Decision Diagram (GPDD) representation explained in Section II. In Section III a symbol reordering strategy is presented for more compact GPDD representation when multiple elements are inserted successively. Preliminary experimental results are presented in Section IV with a conclusion made in Section V.

II. Incremental Construction Algorithm

A. Review on GPDD

The GPDD algorithm developed in [5] is a result of reformulating the two-graph algorithm [1] by incorporating a BDD-based representation. Taking the advantage of data structure sharing offered by BDD, the tree-pair enumeration required by the two-graph method is reformulated into a form of graph-pair reduction, which processes a linearized circuit element-wise. Given an arbitrary linear circuit element, two decisions are made: one substituting the element by a nullor, which is equivalent to letting the element symbol take infinity while the other substituting the element by a *zero* element, whose meaning will be explained. While a pair of graphs is reduced by edge-pairs, all common pairs of sub-graphs are shared and saved in a BDD. During graph-pair reduction, each collapsing of edge-pair is associated with a sign, which is stored with the decision arrows in the GPDD [5].

When all possible graph-pair reduction paths are exhausted, a data structure as illustrated in Fig. 1 will be created, which is called a GPDD. In a GPDD, all passive RLC elements appear in admittance form for manipulation, while all the rest dependent elements have their gains E_k, F_k, G_k, and H_k manipulated as the symbols. Depending on the element types, a set of binary graph-edge reduction rules have been established in [5]. Whenever a pair of graphs is reduced into a pair of single nodes, the reduction path is completed and terminated at the *One* vertex, while all the rest non-completed reduction paths are terminated at the *Zero* vertex. By scanning a path from the GPDD root to the terminal vertex *One*, a symbolic product terms is generated by collecting all symbols and signs encountered along the path, excluding those symbols from which a *dashed* arrow emanates. More details on the GPDD data structure can be found in [5].

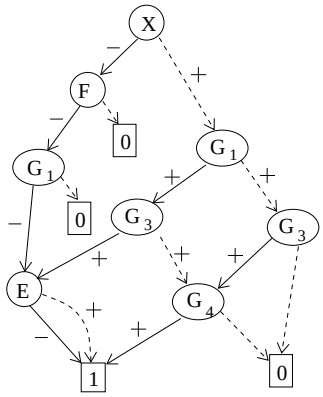

Fig. 1. GPDD example.

*This research was supported by the Natural Science Foundation of China (NSFC Grant No. 61176129).

A GPDD is a bottom-up recursive computation data structure that performs *multiplication* by a *solid* and *addition* by a *dashed* arrow. The computation starts from the two terminal vertices (*One* and *Zero*), and steps upward until the root is reached. The input-output (I/O) transfer function of a circuit is symbolized by a dependent source in GPDD and represented by a special symbol X, which always appears at the GPDD root. The GPDD recursion would generate a sum-of-product (SOP) expression at the GPDD root. For the example GPDD shown in Fig. 1 we get the following *signed* SOP expression

$$XFG_1(E-1) - G_1G_3(E-1) + G_3G_4 = 0, \qquad (1)$$

which must be equal to zero according to the theory established in [5]. The terms in this expression can be divided into two parts: those multiplied by X and those not. This fact has an interesting circuit interpretation which does not come with other symbolic methods. Those terms multiplied by X is actually an SOP representation for the circuit when the X element is replaced by a *nullor*. On the other hand, those terms not multiplied by X is an SOP for the circuit with the X element set to zero. The detailed branch operation for making an element zero depends on the element type, which will be explained a little later. A symbolic expression for $1/X$ is then derived by dividing these two parts [5].

The above statement leads to the following proposition:

Proposition 1: Suppose a circuit has an arbitrary two-port element K of type E, F, G, or H, which might degenerate to one-port. The symbolic product terms of this circuit involve two parts: all terms in one part are multiplied by K while all terms in the rest are not multiplied by K. Those terms in the first part (after removing K) can be generated from the original circuit by replacing the K element by a nullor. Those terms in the second part can be generated by setting the K element to zero.

By setting an element K to zero, it means that the branches associated with the element in the circuit are operated in accordance to the element type: E_0 (*open VC; short VS*), F_0 (*short CC, open CS*), G_0 (*open VC; open CS*), and H_0 (*short CC; short VS*), where the subscript '0' indicates the zero element of that element type. The listed branch operations are justified in [5].

We point out that setting a symbol value to *infinity* or *zero* can be used to alter the circuit topology, whereas the corresponding symbolic expression can be derived simply by slightly modifying the existing GPDD structure. This observation is fundamental to our incremental construction algorithm.

Suppose a GPDD has been created involving symbol K. Multiple GPDD vertices could be associated to the same symbol K (see Fig. 1). For convenience we use subscript indexed symbols to differentiate the multiple vertices such as $K_i, i = 1, 2, \cdots, m_K$, where m_K is the multiplicity of symbol K.

If we let $K = \infty$ (called *infinite symbol*), the GPDD can be simplified by keeping the *solid* arrow emanating from all K vertices while terminating at zero all the *dashed* arrows emanating from the vertices K_i. In case the symbol K is missing in a path, the path must be terminated by a solid arrow to zero as well. Similarly, if $K = 0$ (called *zero symbol*), the GPDD can be simplified by keeping the *dashed* arrow emanating from the K vertices while terminating at zero all the *solid* arrows from the vertices K_i. In case the symbol K is missing in a path, that path remains unchanged. The cost of such operations is no more than $O(|GPDD|)$, where $|GPDD|$ denotes the size of a created GPDD.

The incremental symbolic construction algorithm is intuitively a reversed process of the above GPDD simplification procedure.

B. Incremental Algorithm

We shall develop an element insertion procedure in this section by considering the insertion of a single G-type element first. The insertions of other element types can be performed analogously.

Suppose we would like to insert an element G into a circuit at a selected port (a, b). The port (a, b) could be existing or created by tearing a node apart. The insertion procedure is motivated by the following intuitive discussion: Suppose the element G exists in the circuit and a GPDD has been created with the G-symbol ordered the first, the I/O symbol X the second, and so on. Shown in Fig. 2 is the top part of such a GPDD.

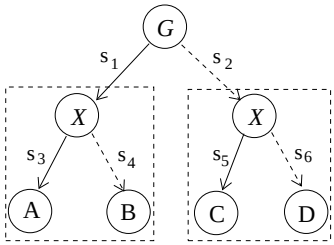

Fig. 2. GPDD with the symbol G at the root. The s_k's are signs.

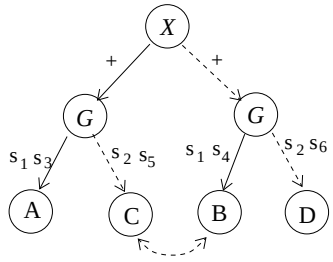

Fig. 3. GPDD with the symbol X moved to the root after swapping G and X.

Let the symbolic function at the root G be f. We denote the two functions at the two X vertices $f|_{G=\infty}$ and $f|_{G=0}$ respectively for X pointed by the solid arrow emanating from G and the dashed arrow from G. We consider the following two cases: (i) If G is inserted by tearing a node, then the reduced circuit associated to $f|_{G=\infty}$ becomes the original circuit because the split apart nodes are merged again. (ii) If G is inserted by connecting to a port (a, b), then the reduced

circuit at $f_{G=0}$ becomes the original circuit because the port is restored.

Thinking reversely, if a GPDD for a circuit has been created before the insertion of G. Then, if G is inserted by tearing a node apart, then the subGPDD at $f|_{G=\infty}$ (see the left framed part in Fig. 2) can be reused (i.e., need not be reconstructed). We only need to construct the other subGPDD at $f_{G=0}$ (see the right framed part in Fig. 2), which corresponds to a circuit with the selected node torn apart (open port). The second case for inserting G at an existing port $(a,\ b)$ is symmetric to the first case in that the right framed subGPDD in Fig. 2 can be reused while the left framed subGPDD needs to be newly created with the selected port shorted.

By constructing the complementary subGPDD as mentioned above, the GPDD for the increased circuit can be assembled as shown in Fig. 2 with the symbol G still at the root. If we want to move the X symbol to the root (for the convenience in GPDD evaluation etc.), we may swap the symbols G and X to get a GPDD shown in Fig. 3. Along with the swapping, the arrow signs have to be modified accordingly meanwhile the two vertices marked "B" and "C" are switched. It is easy to verify that the two GPDDs shown in Figs. 2 and 3 are equivalent.

The incremental symbolic construction algorithm is now summarized.

Incremental GPDD (iGPDD) Algorithm:

Input: A circuit *ckt* with specified I/O (denoted by symbol X). Select a port to insert a new element K in *ckt* without changing the I/O. Let the *ckt* be represented by graph $\mathcal{G}$. Suppose a GPDD for the graph $\mathcal{G}$ has been created.

Output: A new GPDD for *ckt* with the new element K inserted.

Step 1. Create a graph $\mathcal{G}_\infty$ for the case where the insertion port is connected by a nullor; create another graph $\mathcal{G}_0$ for the case where the port is substituted by the $zero$-element of E, F, G, or H-type. (Note that connecting a nullor to a one-port is equivalent to short-circuiting that port.)

Step 2. Check whether $\mathcal{G}_\infty$ or $\mathcal{G}_0$ is identical to the existing graph $\mathcal{G}$. If identical, reuse the existing GPDD for composing the new GPDD.

Step 3. Create a GPDD for the graph $\mathcal{G}_\infty$ or $\mathcal{G}_0$ unequal to $\mathcal{G}$ while using the existing hash table for sharing. In case both $\mathcal{G}_\infty$ and $\mathcal{G}_0$ are unequal to $\mathcal{G}$, we create two GPDDs for them by sharing the existing hash table.

Step 4. Suppose the element K has been inserted in the *ckt*. Reduce the graph once by the element K to fix the two arrow signs s_1 and s_2 in Fig. 2 (with G replaced by K). Keep the rest of the GPDD arrow signs emanating from X's.

Step 5. Swap the symbols K and X as shown in Fig. 3 (with G replaced by K) by modifying the signs accordingly.

If multiple circuit elements need to be added to a circuit, the above *iGPDD* algorithm is invoked for multiple times.

III. Symbol Reordering

In our initial implementation of the *iGPDD* algorithm, we always place the newly inserted symbol at the GPDD root, which results in a symbol order specified by the insertion order. Such an arbitrary order might lead to a large GPDD size, because the GPDD size in general is highly sensitive to the chosen symbol order. In practice, it is suggested to adopt a heuristic order as recommended in [5]. In the initial version of iGPDD we adopted a strategy called *symbol order sifting* whenever a new element is inserted.

Suppose W_1 and W_2 are two neighboring symbols in a GPDD. Swapping the symbols W_1 and W_2 can be written by the following arithmetic expressions which hold trivially (assuming all signs are absorbed by the $f()$ functions)

$$\begin{aligned} f_{top} &= W_1[W_2 f(A) + \boldsymbol{f(B)}] + [W_2 \boldsymbol{f(C)} + f(D)] \\ &= W_2[W_1 f(A) + \boldsymbol{f(C)}] + [W_1 \boldsymbol{f(B)} + f(D)]. \end{aligned} \tag{2}$$

We see clearly that the factors of $f(B)$ and $f(C)$ have been switched in the above two expressions.

Two special cases must be considered for the sake of implementation. Due to the *zero-suppression* in GPDD compaction, some vertices connected to *zero* by solid arrows can be suppressed (removed) from GPDD [5]. The two cases are illustrated in Fig 4, whose equivalences are obvious.

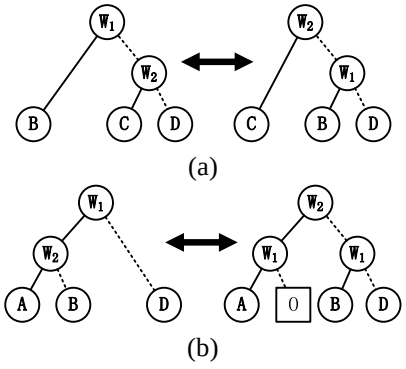

Fig. 4. Special cases of symbol swapping: (a) Case 1: W_2 is zero-suppressed following the solid arrow from W_1 before swapping. (b) Case 2: W_2 is zero-suppressed following the dashed arrow from W_1 before swapping.

We point out that swapping two neighboring symbols only affects the GPDD connection between two neighboring layers with the swapped symbols without changing the rest GPDD structure. Hence, the runtime overhead of swapping is not serious.

Successively swapping multiple symbols is called *sifting*, a procedure proposed by Rudell for ordered BDD implementation [8]. The basic idea of sifting goes as follows: Pick a symbol in BDD, sift it downward or upward until a relatively small BDD results.

Since in *iGPDD* we always place the newly inserted symbol at the GPDD root, only downward sifting is necessary until a relatively small GPDD is obtained.

IV. Experimental Results

A simple RC integrator is shown in Fig 5 (on the left) together with a macromodel for the operational amplifier (opamp) (on the right). We successively add the circuit elements in the macromodel to test the efficiency of incremental construction. The macromodel elements are the finite gain E_{gain}, the buffer gain E_{buf}, the input and output resistances R_{in} and R_{out}, and R_{p1} and C_{p1} defining a pole, They are inserted incrementally in the written sequence.

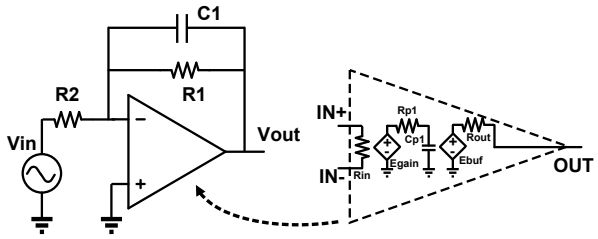

Fig. 5. RC integrator for testing the incremental algorithm.

The CPU time for insertion and the GPDD size growth are listed in Table I. The insertion of each symbol takes a small amount of time, but the time grows as more symbols are inserted. The accumulated incremental construction time is 1,522 μs. For comparison, we also measured the non-incremental construction time by creating a GPDD including all macromodel symbols with the same symbol order; it took 1,736 μs with the GPDD size 21. Because this circuit is small, the incremental construction outperforms the non-incremental construction. For large circuits, a complete run of all incremental constructions for inserting a set of N elements would normally be slower than one run of GPDD construction by including the N elements altogether, but faster than N runs of restarted GPDD constructions.

TABLE I
INCREMENTAL INSERTION RESULTS FOR THE RC INTEGRATOR.

Inserted symbol	Insertion time (μs)	GPDD size
(Starting circuit)	–	5
E_{gain}	34	8
E_{buf}	44	12
R_{in}	54	14
R_{out}	119	19
R_{p1}	36	22
C_{p1}	259	27

Next we tested the incremental construction with sifting. By sifting, every inserted symbol is rearranged to a better position downward in the symbol list. A bandpass filter shown in Fig 6 is used for this experiment, in which we modified the three opamp models by inserting additional macromodel elements: first inserting both E_{gain} and E_{buf}; then inserting both R_{in} and R_{out}; and finally inserting both R_{p1} and C_{p1} to all three opamps.

Listed in Table II is a comparison of three different test settings. The first row shows the GPDD size and construction time for the whole circuit including all macromodel elements (i.e., non-incremental construction). The symbol order came from a heuristic ordering. This result is used for comparison to the other two tests listed in the next two rows. The incremental construction without sifting (the 2nd row) takes less time than that with sifting (the 3rd row), but the GPDD size is much larger as expected.

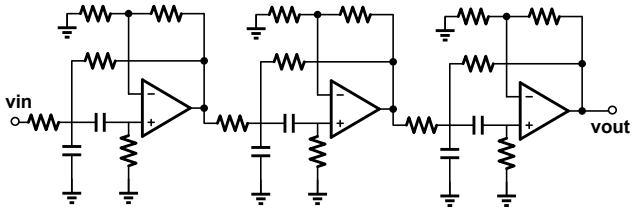

Fig. 6. The bandpass filter containing three opamps for the second test circuit.

TABLE II
COMPARISON OF INCREMENTAL CONSTRUCTION WITH SIFTING.

Test case	GPDD size	Total time (ms)
Non-incremental w. pre-order	160	31.00
Incremental w/o sifting	2141	1042.70
Incremental with sifting	326	1334.30

V. Conclusion

An incremental symbolic method for tracing analog circuit topology modification has been proposed. The algorithm has been developed as a simple extension of the existing GPDD algorithm developed for topological analog network analysis. By taking the advantage of a distinguished property owned by GPDD, namely, all circuit elements are directly manipulated as symbols in GPDD, modifying an already constructed GPDD data structure to trace incremental circuit topology change is a feasible technique. In the future we shall investigate other better insertion strategies by approaches such as modular insertion to improve the efficiency and interactiveness.

References

[1] P. M. Lin, *Symbolic Network Analysis*. New York: Elsevier, 1991.

[2] F. V. Fernández, A. Rodríguez-Vázquez, J. L. Huertas, and G. Gielen, Eds., *Symbolic Analysis Techniques – Applications to Analog Design Automation*. New York: IEEE Press, 1998.

[3] M. Fakhfakh, E. Tlelo-Cuautle, and F. V. Fernández, Eds., *Design of Analog Circuits through Symbolic Analysis*. Oak Park, IL, USA: Bentham Science Publishers (e-Books), 2012.

[4] C. J. R. 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, January 2000.

[5] G. Shi, "Graph-pair decision diagram construction for topological symbolic circuit analysis," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, no. 2, pp. 275–288, February 2013.

[6] ——, "A survey on binary decision diagram approaches to symbolic analysis of analog integrated circuits," *Analog Integrated Circuits and Signal Processing*, vol. 74, no. 2, pp. 331–343, 2013.

[7] C. Ferent and A. Doboli, "Symbolic matching and constraint generation for systematic comparison of analog circuits," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, no. 4, pp. 616–629, 2013.

[8] R. Rudell, "Dynamic variable ordering for ordered binary decision diagrams," in *Proc. ACM/IEEE Int'l Conf. on Computer-Aided Design (ICCAD)*, 1993, pp. 42–47.