Algorithms for Taylor Expansion Diagrams*

Görschwin Fey Rolf Drechsler
Institute of Computer Science
University of Bremen
28359 Bremen, Germany
{fey,drechsle}@informatik.uni-bremen.de

Maciej Ciesielski
Department of Electrical Engineering
University of Massachusetts
Amherst, MA 01003, USA
ciesiel@ecs.umass.edu

Abstract

The ever increasing size of integrated circuits results in large problem sizes during synthesis and verification of such designs. Recently Taylor Expansion Diagrams (TEDs) were introduced as a data structure to cope with large problem instances. TEDs allow to exploit high level information in the representation of functions.

In this paper the basic TED operations are analyzed from a complexity point of view. Suggestions for optimizations of the originally proposed algorithms are made.

1. Introduction

The size and complexity of modern circuit designs are rapidly growing. This results in the demand for data structures to handle large problem instances in synthesis and verification. *Binary Decision Diagrams* (BDDs) boosted the development of tools in the 1990s, but are restricted by their binary decomposition scheme [2]. Other types of decision diagrams were introduced to increase the efficiency. Various extensions were investigated, including FDDs [13, 9] or KFDDs [8], and word-level representations such as EVBDDs [14], FEVBDDs [17], *BMD [3], HDDs [6] or K*BMDs [7]. But these types of decision diagrams also suffer from complexity problems (see e.g. [10, 1, 15]). In particular exponential bounds for synthesis operations were proven for decision diagrams based on Davio decomposition, while these operations are polynomial on BDDs.

To overcome these limitations *Taylor Expansion Diagrams* (TEDs) were introduced in [5]. TEDs rely on the inherently non-binary Taylor expansion as the basic decomposition scheme. This allows to efficiently group binary variables and treat them as multi-valued inputs. The size, i.e. the number of nodes, of a TED is smaller for several functions than that of other decision diagrams. TEDs have been successfully applied to the verification of behavioral and algorithmic descriptions [12]. However, the presented results were purely experimental, and worst case computation complexities for the algorithms were not given.

In this paper the complexity of TEDs representing Boolean functions and the complexity of operations performed on integer-valued TEDs are investigated. A first analysis of the time complexity of operations on TEDs was given in [11] and [5]. Here, the analysis is carried out in detail for addition and multiplication of two normalized TEDs. Some modifications to the originally proposed algorithms are suggested. Namely, these are the early extraction of a *Greatest Common Divisor* (*GCD*) and the use of edges for function representation. These modifications can not reduce the asymptotic run time, but the average run time.

The paper is structured as follows: In Section 2 TEDs and necessary notations are formally introduced. The representation of Boolean functions by TEDs is described in Section 3. Addition and multiplication of two TEDs are studied in Section 4 and Section 5, respectively. In both cases the algorithm is introduced at first, then the complexity is analyzed in detail. Summary and conclusions are given in Section 6.

2. Preliminaries

A TED is a representation of a function as a directed acyclic graph. A function is given by an edge, associated with the root node, and is called a *root edge* in the sequel. Using edges besides nodes to denote a function facilitates the factorization of the TEDs, an operation necessary to perform normalization. A node represents a function with factor 1.

Each node v in a TED is either a terminal node or an internal node labeled with a variable, denoted by $v \rightarrow index$. A terminal node has an associated value, the maximum index is reserved to mark a node as terminal. At each internal node v the Taylor expansion with respect to variable $x = v \rightarrow index$ at position $x = 0$ is applied to decom-

* This work was supported in part by DAAD grant D/0247216 and in part by DFG grant DR 287/8-1.

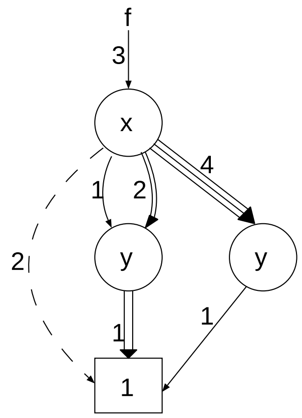

Figure 1. TED representing $f = 6 + 3xy^2 + 6x^2y^2 + 12x^3y$ for the ordering x, y

pose the represented function:

$$f(x) = f(0) + xf'(0) + x^2\frac{1}{2}f''(0) + \ldots$$

An outgoing edge e of v points to a derivative $f^k(x)$ evaluated at $x = 0$. Edge e has the order $e \rightarrow order = k$ and an associated constant multiplicative value $e \rightarrow weight$. Edges with weight 0 are removed. Given an edge $e = (v, w)$, the node v is called source, w is called target. $|F|$ denotes the number of nodes in the TED F. A TED is called *reduced* if no isomorphic subgraphs exist, i.e. subgraphs with the same structure, the same labels of nodes, edge weights and edge orders. It is called *ordered* if variables occur only once and in the same order on all paths to the terminal. A TED is called *normalized*, if the GCD of the weights of all outgoing edges is 1 and only one terminal node with value 1 exists. Reduced ordered normalized TEDs are a canonical representation of functions. Only these are considered in the sequel.

From the definition of TEDs it is clear, that the function to be represented has to be continous. Usually the function is only evaluated at integer-valued points. If the represented function can be expressed as a finite order polynomial, the Taylor expansion of the function has a finite number of terms. Thus, the TED exactly represents the function.

Example 1. *Figure 1 represents $f = 6 + 3xy^2 + 6x^2y^2 + 12x^3y$ with respect to variable order (x, y). 0-order edges are dotted, first order edges are a single line, higher orders are indicated by double and triple lines.*

Similar to efficient BDD-packages [16] a TED-package needs a unique table to guarantee that no isomorphic subgraphs occur. Also a cache of already computed results can reduce the runtime during operations.

Operations on two TEDs are carried out in a recursive manner. Adding two TED-nodes requires the adding of each pair of the outgoing k-order edges. Multiplying two TED-nodes is accomplished by multiplying all pairs of outgoing edges. For simplicity during analysis of the complexity of operations on TEDs *GCD* is treated as a constant time function.

Boolean	numeric
$\overline{x}$	$1 - y$
$x \wedge y$	$x \cdot y$
$x \vee y$	$x + y - xy$
$x \oplus y$	$x + y - 2xy$

Table 1. Modeling Boolean operators in a numeric domain

3. Representing Boolean Functions

Usually TEDs can represent continous functions. When Boolean functions are considered, some modifications need to be made to maintain a canonical data structure. The resulting diagrams, when representing only Boolean functions, are called BTEDs. (Note: TEDs can contain Boolean variables as part of the integer-valued function representation). This section briefly reviews BTEDs and points to their similarity with *BMDs. This similarity allows to apply the results regarding the complexity of *BMDs directly to BTEDs.

To ensure correct evaluation of Boolean expressions the Boolean operators are modeled in the numeric domain using the equations shown in Table 1.

For a Boolean variable $x \in \mathbb{B}$ the identity $x = x^1 = x^2 = \ldots = x^k$ for $k \geq 1$ must be applied to maintain a canonical representation of Boolean functions in TEDs (otherwise e.g. $\overline{x} = 1 - x \neq (1 - x)^2 = \overline{x}\,\overline{x}$). By definition, only linear functions may occur over Boolean variables in BTEDs. In this case the Taylor expansion with respect to a Boolean variable x of a function $f(x)$ becomes:

$$f(x) = f(0) + x(f(1) - f(0))$$

This is identical to the decomposition used in *BMDs [3, 4]. Thus, the BTED and *BMD for Boolean functions are structurally identical. Results on the complexity for *BMDs representing Boolean functions can directly be transferred to BTEDs.

Relations between different types of graphical function representations have been investigated in [1]. The following lemma follows directly from [1] and the above considerations.

Lemma 1. *Let $B = \{f | f : \mathbb{B}^* \rightarrow \mathbb{B}\}$.*

a) *There exist functions in B that have a BDD representation of polynomial size but only a BTED (*BMD) representation of exponential size.*

b) *There exist functions in B that have a BTED (*BMD) representation of polynomial size but only a BDD representation of exponential size.*

c) *The BTED of a function in B is always larger than or equal in size to the FDD of the function.*

Further relations between BTEDs and other diagram types can be derived from [1] as well.

All results stated in this section are valid for Boolean functions only. TEDs can represent arbitrary integer valued functions that can be expressed as polynomials, and can readily include Boolean variables and Boolean subfunctions. Additionally it has been shown in [5] that TEDs are more efficient than *BMDs for non-linear (polynomial) functions.

The next sections investigate the complexity of operations on TEDs in general.

4. Addition

The two TEDs to be added are given by two k-order edges, f and g. At first some aspects of the add operation are considered and the algorithm is introduced. Then, the complexity of the operation is analyzed in detail.

4.1. Algorithm

The algorithm in Figure 2 is in most parts identical to the one given in [11]. Modifications include adding the extraction of the *GCD* in line (3) and the accesses to the computed table in lines (6,38).

The flow of the algorithm is as follows: At first terminal cases are checked (0-2). Then, the computed table is checked for the result (6-8), which is returned upon success (39,40). Otherwise the recursion is carried out on all succeeding edges (9-34). The result is reduced, normalized and inserted into the unique table and computed table (35-38).

```
     TED_ADD(f, g)
(0)  if terminalCase(f, g) then
(1)    return terminalResult(f, g)
(2)  end if
(3)  c = GCD(f→weight, g→weight)
(4)  f→weight = f→weight/c
(5)  g→weight = g→weight/c
(6)  if ComputedTable_Lookup(f, g, h) then
(7)    goto result
(8)  end if
(9)  if g→node→index > f→node→index then
(10)   Exchange f and g
(11) end if
(12) if TED_NodeIsTerminal(f→node) then
(13)   h→node = g→node /* Copy including edges */
(14)   Push g→weight to all children of h
(15)   h→node→edge[0] =
          TED_ADD(f, h→node→edge[0])
(16) else if f→node→index > g→node→index then
(17)   h→node = f→node /* Copy including edges */
(18)   Push f→weight to all children of h
(19)   h→node→edge[0] =
          TED_ADD(g, h→node→edge[0])
(20) end if

(21) if f→node→index == g→node→index then
(22)   if f→node→num_edges < g→node→num_edges then
(23)     Exchange f and g
(24)   end if
(25)   if f→node→num_edges > g→node→num_edges then
(26)     h→node = f→node /* Copy including edges */
(27)     Push f→weight to all children of h
(28)     i→node = g→node /* Copy including edges */
(29)     Push g→weight to all children of i
(30)   end if
(31)   for k = 0 to i→num_edges do
(32)     h→node→edge[k] =
            TED_ADD(h→node→edge[k], i→node→edge[k])
(33)   done
(34) end if

(35) h→node = TED_Reduce(h)
(36) h = TED_Normalize(h)
(37) h→node = TED_FindOrAddNode(h→node)
(38) ComputedTable_Insert(f, g, h)
(39) result: h→weight = h→weight · c
(40) return h
```

Figure 2. Algorithm to add two TEDs f and g

Terminal cases There are two terminal cases for the addition:

1. If an edge weight is zero, one operand is zero. In this case the result is the other operand.
2. Both edges point to identical nodes. Then the edge weights have to be added and the resulting edge points to the same node. The following example shows this case for subfunction $h(y)$ (see Figure 3).

Example 2. *Let $f = f_0 + f_1 h(y)x$ and $g = g_1 h(y)x$. Then $f + g = f_0 + (f_1 + g_1)h(y)x$.*

Propagation of weights As shown by the following example, the propagation of weights is necessary. The weight has to be taken into account when recursively adding the polynomials. But when carrying out the recursive addition the *GCD* of the weights of the two operands can be extracted. This results in smaller weights during recursive calls and in a higher probability to retrieve already computed results from the computed table. The edges passed to the operation have to be copies, in order not to modify the functions represented by the TED when the divisions in lines (4,5) are carried out.

Example 3. *Let $f = f_0 + f_1 x(f_2 + f_3 y)$ and $g = g_0 + g_1 x(g_2 + g_3 y)$.
Then $f+g = (f_0+g_0)+(f_1 f_2+g_1 g_2+(f_1 f_3+g_1 g_3)y)x$.
Now, let $c = GCD(f_1, g_1)$ and $f_1' = f_1/c$, $g_1' = g_1/c$.
Thus $f+g = (f_0+g_0)+cx(f_1' f_2+g_1' g_2+(f_1' f_3+g_1' g_3)y)$.*

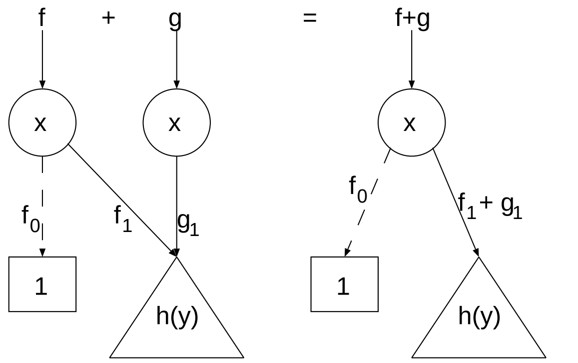

Figure 3. Addition of TEDs F and G that have an identical subgraph

Computed table For BDD-operations the computed table guarantees, that no two nodes have to be processed twice. When TED_ADD is applied to two root edges the associated weights have to be taken into account. Only if the target nodes have been processed with these weights already, the previous result can be returned. Otherwise the nodes have to be processed again.

4.2. Complexity

The number of calls to TED_ADD is bounded by the maximum order k of the represented polynomials and the number of variables n.

The basic idea of the following "worst-case-example" is that all weights have to be propagated, if no *GCD* for any two edges can be found. From this it follows that a lookup in the computed table will never succeed, since no particular weight occurs twice with respect to a node. Therefore all paths in the TED have to be enumerated.

Example 4. *Consider the TED F given in Figure 4. Let G be a structural identical TED having weights $g_{l,m}$. Let all the weights $f_{i,j}$ and $g_{l,m}$ be prime numbers and let no prime number occur twice. When TED_ADD is applied to F and G, all paths are iterated by a depth first traversal from top to bottom. F and G have the same structure and do not share nodes. Therefore no terminal cases occur besides reaching the terminal in both TEDs at the same time.*

Furthermore, none of the previously computed results in the computed table stops the recursion which is due to the following: Each time a node s in F is visited, there is a corresponding node t in G. The same is true for paths p and q in F and G, respectively. The edge weights are pairwise different primes on the corresponding paths. Therefore, when reaching a node s in the TED F along a path p from the root, all the weights along this path have been successively multiplied. Let the accumulated weights along paths p and q be $w_{s,p}$ and $w_{t,q}$, respectively. Since the current path p differs from any other path p' during a previous visit in at least one edge there is at least one prime number in $w_{s,p}$ that is not in $w_{s,p'}$ and therefore the weight $w_{s,p}$ did not occur previously. The same is true for weights at node t.

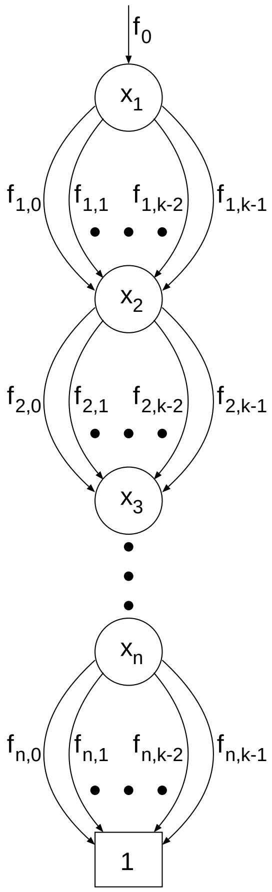

Figure 4. TED F, all $f_{i,j}$ are prime and no prime number occurs twice

Therefore the number of calls to TED_ADD with respect to an edge is equal to the number of paths arriving at the source node of this edge from the root edge. At the node labeled x_i the number of arriving paths is k^{i-1} and the operator is called for each of the k outgoing edges of the node, i.e. k^i times. Additionally the operator is called for the root edge. The total number of calls to TED_ADD is the sum of calls to any edge:

$$\sum_{i=0}^{i=n} k^i = \frac{k^{n+1}-1}{k-1} = \mathcal{O}(k^n)$$

Theorem 1. *Let F and G be two TEDs representing functions f and g of n variables. The highest order term in either function is no greater than $k-1$. The number of calls to the ADD-operation is bounded by $\mathcal{O}(k^n)$. This bound is tight.*

Proof. Example 4 shows that this bound may be reached.

That the number of calls to ADD is in $\mathcal{O}(k^n)$ is shown in the following. First consider the case, that all n variables occur on any path within F and G and any node has k outgoing edges. The number of paths arriving at a node v from the root edge is the upper bound on ADD-calls to edges having this node as target. This is the case in Example 4. For weights different to the primes from the example the number of calls may only decrease, because of a successful lookup in the computed table.

Now, assume that any node has a fanout of less than k, i.e., the higher order edges disappear. These edge weights are considered as being zero and no recursion is performed on these edges, the number of calls to ADD decreases.

Finally, consider the case when not all the variables occur on all paths. If identical variables are missing on corresponding paths in F and G the operation directly jumps to the next level and the number of possible paths decreases. If the variable is only missing in G, the recursive call is only done for the 0-child of the node in F and, again, the number of paths decreases. ■

This shows that optimizations of the ADD-operation like *GCD*-extraction and computed table can not prevent the exponential behavior. In the following section the exponential worst case behavior of multiplication is referred to that of addition.

5. Multiplication

In this section the multiplication of two TEDs is investigated in more detail. At first the algorithm is outlined. Then the complexity is analyzed.

5.1. Algorithm

For brevity no algorithm for the multiplication of two functions given as TEDs is shown. The basic algorithm works as the ADD-algorithm shown in the previous section. The differences are as follows: a) the terminal cases are different; b) the computed table can be more efficient; and c) the recursion is more complex.

Terminal Cases Only two terminal cases occur:

1. One operand is an edge pointing to a terminal. Then, the other operand is multiplied by a constant.
2. One operand is zero. The result is zero.

Computed Table The computed table can reduce the number of calls to TED_MULT more efficiently than in the case of addition. Weights can be factored out, i.e. a pair of nodes has only to be processed once. Thus, the lookup table only contains the targets of edges, i.e. nodes, but no edges and weights.

Example 5. *Consider functions $f(x)$ and $g(x)$. Then to calculate $a_1 f \cdot b_1 g$ and $a_2 f \cdot b_2 g$, the product $f(x)g(x)$ has only to be calculated once. The different weights $a_1 b_1$ and $a_2 b_2$ are factored out and are contained in the edge pointing to the result of $f(x)g(x)$.*

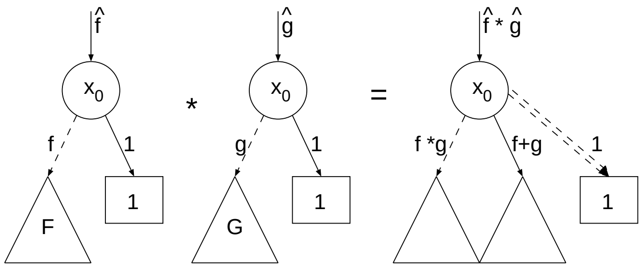

Figure 5. Multiplication of two TEDs

Recursion During addition only pairs of successors of the same grade have to be considered. For multiplication any pair of successors has to be considered and partial sums have to be calculated.

Example 6. *Consider functions f and g:*

$$\begin{aligned} f(x) = &\quad a_f f_{1,0} f(0) + b_f f'(0)x + c_f f''(0)x^2 \\ g(x) = &\quad a_g g_{1,0} g(0) + b_g g'(0)x + c_g g''(0)x^2 \end{aligned}$$

The product $f(x) \cdot g(x)$ contains terms of orders up to x^4. E.g. the following term of order 2:

$$(a_f c_g f(0)g''(0) + c_f a_g f''(0)g(0) + b_f b_g f'(0)g'(0))x^2$$

The intermediate additions may result in propagation of weights as shown by the following analysis.

5.2. Complexity

The number of calls to TED_MULT can be efficiently reduced by using a computed table to store results.

Theorem 2. *Let F and G be two TEDs representing polynomials with highest order k. Then the number of calls to TED_MULT when calculating $F \cdot G$ is bound by $\mathcal{O}(k^2 \cdot |F| \cdot |G|)$.*

Proof. No further recursive calls of TED_MULT occur, if the pair of operands is found in the computed table. Upon the first visit of a pair all children have to be multiplied pairwise resulting in at most $(k+1)^2$ calls to MULT. ■

Nonetheless the calculation of the partial sums can cause an exponential worst case behavior. The scheme to calculate the product $h(x_i, \ldots, x_n)$ of two functions $f(x_i, \ldots, x_n)$

and $g(x_i, \ldots, x_n)$ is given by the following equation:

$$\begin{array}{lll} h(x_i,\ldots,x_n) = & & \\ & [& f(0,x_{i+1},\ldots,x_n)g(0,x_{i+1},\ldots,x_n)] \\ + & [& f(0,x_{i+1},\ldots,x_n)g^{(1)}(0,x_{i+1},\ldots,x_n) \\ & + & f^{(1)}(0,x_{i+1},\ldots,x_n)g(0,x_{i+1},\ldots,x_n) \\ &] & x \\ + & \ldots & \\ + & [& f^{(k)}(0,x_{i+1},\ldots,x_n)g^{(k)}(0,x_{i+1},\ldots,x_n) \\ &] & x^{2k} \end{array} \quad (1)$$

The sums of partial products are denoted by square brackets.

Theorem 3. *All algorithms using the recursive scheme of Equation 1 to multiply two TEDs are exponential in the worst case.*

Proof. Consider the multiplication shown in Figure 5. Given are functions $\hat{f}(x_0,\ldots,x_n) = f(x_1,\ldots,x_n) + x_0$ and $\hat{g}(x_0,\ldots,x_n) = g(x_1,\ldots,x_n) + x_0$. Let subfunctions f and g be defined as in Example 4 and Figure 4. The result of the multiplication is $\hat{f} \cdot \hat{g} = f \cdot g + (f+g)x_0 + x_0^2$.

Therefore $f + g$ has to be calculated which involves an exponential number of calls to TED_ADD: At first assume that during calculation of $f \cdot g$ none of the calls necessary to calculate $f + g$ are carried out. Then, the exponential behavior results from the previous considerations. Second, assume that TED_ADD has been called earlier with respect to two arguments in the representation of f and g while $f \cdot g$ is calculated. With respect to these arguments the number of calls to TED_ADD does not decrease, because the computed table does not contain the necessary results for the same reasons as discussed previously in Example 4 and the proof of Theorem 1. In summary at least the same number of calls to TED_ADD as in Example 4 is needed. ■

Thus, both operations addition and multiplication exhibit exponential worst case behavior when applied to TEDs. In practical applications the occurring operations can be expected to exhibit a more "friendly" behavior, as all factors to functions are derived from the same circuit. This hypothesis is supported by the experiments given in [12].

6. Conclusions

The complexity of TED representations and operations was theoretically analyzed. For the representation of Boolean functions results were retrieved by showing the identity between TEDs and *BMDs in that case. Addition and multiplication were investigated for the general case and proven to have exponential worst case complexity. Examples that exhibit this worst case were presented.

It is focus of current work to implement an efficient TED package and to further investigate the complexity of operations for circuits that occur in practice.

References

[1] B. Becker, R. Drechsler, and R. Enders. On the computational power of bit-level and word-level decision diagrams. In *ASP Design Automation Conf.*, pages 461–467, 1997.

[2] R.E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. on Comp.*, 35(8):677–691, 1986.

[3] R.E. Bryant and Y.-A. Chen. Verification of arithmetic functions with binary moment diagrams. In *Design Automation Conf.*, pages 535–541, 1995.

[4] R.E. Bryant and Y.-A. Chen. Verification of arithmetic circuits using binary moment diagrams. *Software Tools for Technology Transfer*, 3(2):137–155, 2001.

[5] M. Ciesielski, P. Kalla, Z. Zheng, and B. Rouzyere. Taylor expansion diagrams: A compact canonical representation with applications to symbolic verification. In *Design, Automation and Test in Europe*, pages 285–289, 2002.

[6] E.M. Clarke, M. Fujita, and X. Zhao. Application of multi-terminal binary decision diagrams. *IFIP WG 10.5 Workshop on Applications of the Reed-Muller Expansion in Circuit Design*, pages 21–27, 1995.

[7] R. Drechsler, B. Becker, and S. Ruppertz. K*BMDs: A new data structure for verification. In *European Design & Test Conf.*, pages 2–8, 1996.

[8] R. Drechsler, A. Sarabi, M. Theobald, B. Becker, and M.A. Perkowski. Efficient representation and manipulation of switching functions based on ordered Kronecker functional decision diagrams. In *Design Automation Conf.*, pages 415–419, 1994.

[9] R. Drechsler, M. Theobald, and B. Becker. Fast OFDD based minimization of fixed polarity Reed-Muller expressions. In *European Design Automation Conf.*, pages 2–7, 1994.

[10] R. Enders. Note on the complexity of binary moment diagram representations. *IFIP WG 10.5 Workshop on Applications of the Reed-Muller Expansion in Circuit Design*, pages 191–197, 1995.

[11] P. Kalla. *An Infrastructure for RTL Validation and Verification*. PhD thesis, University of Massachusetts, Amherst (MA), September 2002.

[12] P. Kalla, M. Ciesielski, E. Boutillon, and E. Martin. High-level design verification using taylor expansion diagrams: First results. In *Int'l Workshop on High Level Design Validation and Test*, pages 13–17, 2002.

[13] U. Kebschull, E. Schubert, and W. Rosenstiel. Multilevel logic synthesis based on functional decision diagrams. In *European Conf. on Design Automation*, pages 43–47, 1992.

[14] Y.-T. Lai and S. Sastry. Edge-valued binary decision diagrams for multi-level hierarchical verification. In *Design Automation Conf.*, pages 608–613, 1992.

[15] C. Scholl, B. Becker, and T. Weis. On WLCDs and the complexity of word-level decision diagrams — a lower bound for division. *Formal Methods in System Design: An International Journal*, 20(3), 2002.

[16] F. Somenzi. Efficient manipulation of decision diagrams. *Software Tools for Technology Transfer*, 3(2):171–181, 2001.

[17] P. Tafertshofer and M. Pedram. Factored edge-valued binary decision diagrams. *Formal Methods in System Design: An International Journal*, 10(2):243–270, 1997.