Binary Taylor Diagrams (BTD): An Efficient Way of Implementing Taylor Expansion Diagrams

A. Hooshmand, S. Shamshiri, M. Alisafaee, B. Alizadeh, P. Lotfikamran, M. Naderi and Z. Navabi

Department of Electrical and Computer Engineering
University of Tehran, Iran
{arash, shamshiri, alisafaee, bijan, plotfi, mnaderi}@cad.ece.ut.ac.ir, navabi@ece.neu.edu

Abstract

This paper presents an efficient way of implementing Taylor expansion Diagrams. Binary Taylor Diagram (BTD) is based on Taylor series like TED, but uses a binary data structure. It is also compatible with structure of conventional methods which makes it suitable for checking equivalency between two circuits. Complexity of algorithms of functions of BTD is simpler than those of TED; therefore applying operations to BTD take less CPU time.

Index Terms

Binary Taylor Diagram (BTD), High Level Verification, Canonical Representation, Equivalence Checking, Binary Data Structures

1. Introduction

Today hardware complexity increases fast and makes testing process more essential and critical. Validating of a design is better done as fast as possible because the cost of testing increases from core level to chip level and from chip level to system level by an order of magnitude. Therefore, a reliable and high level method for testing design is essential.

Verification is a reliable and high level method that was proposed and improved in the last decade and is a suitable complement for conventional simulation. Simulation is inefficient for large designs, because it is not possible to simulate a large circuit for all possible valid input values in a reasonable time. Verification solves the problems of simulation by applying mathematical rules for proving the validation of a design implementation.

Two types of verification methods are equivalence checking and model checking. For equivalence checking, boolean and arithmetic functions and relations should be represented in a data structure for applying reduction rules, functions and operations.

Till now, several data structures for this representation like *ROBDD* [1], *FDD* [2], *KFDD* [3], *HDD* [4], *BMD* [5], *k*BMD* [6] and *TED* [7] [8] [9] are proposed. All of these methods are graph based and canonical.

Reduced Ordered Binary Decision Diagram (*ROBDD*) was the first canonical graph based representation of boolean functions that was introduced by Bryant [2]. *ROBDD* is a directed binary tree which works with Shannon's decomposition:

$$f^{v}(x_1, x_2, ..., x_n) = [\neg x_i \wedge f^{low(v)}(x_1, x_2, ..., x_n)] \vee [x_i \wedge f^{high(v)}(x_1, x_2, ..., x_n)]$$

where

$$f^{low(v)}(x_1, x_2, ..., x_n) = f(x_1, x_2, ..., x_n)|_{x_i = 0}$$

$$f^{high(v)}(x_1, x_2, ..., x_n) = f(x_1, x_2, ..., x_n)|_{x_i = 1}$$

and var (v) = x_i.

For supporting not only Boolean functions but also arithmetic ones, Word Level Decision Diagrams (WLDD) was introduced. WLDDs work with integer values instead of boolean values and use arithmetic operations (add, mul) beside boolean operations (and, or, not). In this way, *MTBDD* [10], *BMD*, *K*BMD* and *TED* are developed.

The last proposed representation, *TED* (Taylor Expansion Diagram) uses Taylor series as its decomposition method and supports more than one degree arithmetic functions, but unlike the other methods, *TED* is based on a non binary tree. The number of children of a node in *TED* depends on the corresponding variable degree. Therefore, *TED* uses a more complicated data structure than the others.

In the last decades, several efficient algorithms for representing binary data structures (i.e. *ROBDD*) have been proposed [11] [12] [13]. Our proposed representation, i.e. *BTD*, unlike *TED*, uses simple binary data structure. Therefore, it is possible to use these algorithms for implementing an efficient *BTD* package.

Section 2 describes the method of constructing *BTD*, Section 3 shows how different operations are applied to *BTD*, and then Section 4 gives some reduction rules. In Section 5, we prove why *BTD* is canonical and some experimental results are shown in Section 6. The last section gives a short conclusion.

2. Binary Taylor Diagram (BTD)

BTD is a binary tree based representation of arithmetic and boolean functions that uses the Taylor series as its decomposition method (i.e. *TED*).

2.1. Taylor series

The Taylor series is capable to represent all kind of differentiable functions including logarithmic, exponential, and sinusoidal. Therefore Taylor based decomposition allows us to represent designs at high level of abstraction. For example a logarithmic function in the behavioral code could be represented by the finite sentences of the Taylor series with acceptable approximation. In the structural level all the design aspects are represented with polynomial functions. For simplicity, we consider only polynomial functions in our examples and comments. The coefficients of the polynomial could be real numbers in general, but in the most actual designs the integer coefficients are enough for the representation.

The Taylor series of a real differentiable function *f(x)* around *x=0* are:

$$f(x) = f(0) + xf'(0) + \frac{1}{2!}x^2 f''(0) + \frac{1}{3!}x^3 f^{(3)}(0) + \ldots \quad \textbf{(1)}$$

Where *f'(0)*, *f''(0)* and $f^{(3)}(0)$ are derivations from degree one, two and three of the function *f* around *x=0* point respectively.

By factoring *x*, Equation (1) is written as follows:

$$f(x) = f(0) + x(f'(0) + x(\frac{1}{2!}f''(0) + x(\frac{1}{3!}f^{(3)}(0) + \ldots))) \quad \textbf{(2)}$$

In the above equation, the outer bracket is the derivation of function f(x) as follows:

$$f(x) = f(0) + x.f'(x) \quad \textbf{(3)}$$

The decomposition will be performed based on Equation (2) recursively, and each node of *BTD* is decomposed according to the Equation (3).

2.2. BTD construction

BTD is a representation for multi-variable polynomial functions, boolean functions and Taylor expansion series of algebraic functions. *BTD* is a directed weighted binary tree which all of its nodes have a label that identifies a decomposing variable. First, we consider polynomial forms and Taylor series and then we will discuss boolean functions.

Like decision diagrams, *BTD* variables are ordered. Variable ordering is a basic rule to create a canonical form.

For representing the function *f(x)* as a *BTD*, Equation (3) is applied to the root node of the function. The root node has two children, i.e., *f(0)* as the left child and *f'(x)* as the right child. All internal nodes also have two children. The left child is computed by inserting value 0 instead of the decomposing variable in the parent function. The right child is computed by one level derivation of the parent at decomposing variable (see Figure 1). The out-degree of each node is 2 but the out-degree of a terminal node is 0.

There are only two valid terminal nodes, **0** and **1**. All other non-zero terminals could be replaced by a 1 terminal and a multiplicative weight on the incoming edge. The G.C.D. of outgoing edges' weight of each node is sent to the incoming edge for canonicity; therefore the weights of outgoing edges are relatively prime. All **1** terminals merge with each other to reduce the size of *BTD*, and the same is true for the **0** terminals.

To obtain the arithmetic function from the *BTD*, each path from the root to a **1** terminal node should be considered as a term in the function. This term is computed by production of the weights on all edges in the path. Notice that all right edges (1-edges) implicitly have related decomposing variables in their weights.

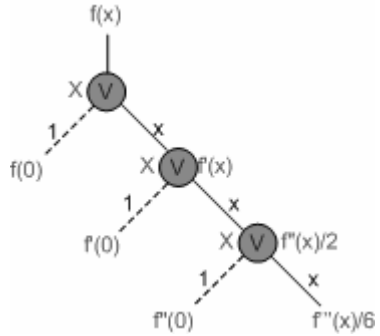


Figure 1. Decomposition node in *BTD*

Example 1: Figure 2 illustrates the *BTD* representation for a simple arithmetic expression, i.e. A^2B+AB.

Example 2: The representation of the $X^2 = (4x_2 + 2x_1 + x_0)^2$ is shown in Figure 3, where *X* is a three bit integer number with boolean digits x_2, x_1 and x_0 (their order is $x_2 > x_1 > x_0$). As this figure shows, the terms

$(4x_2)^2 + 2(4x_2)(2x_1 + x_0)$ and $(2x_1 + x_0)^2$ are considered as the right child and left child of the root respectively.

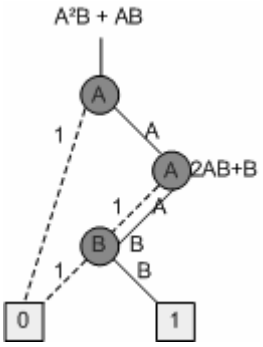


Figure 2. *BTD* representation of function f(A, B) = $A^2B + AB$

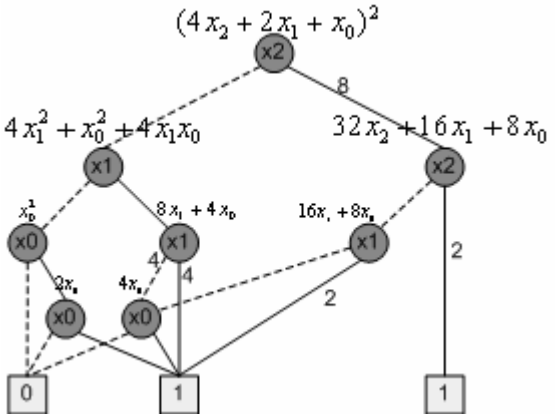


Figure 3. *BTD* representation of function X^K for K = 2, X is an integer value represented by three bits x_2, x_1 and x_0

3. BTD operations

Now we discuss how arithmetic and boolean operations are performed. First, we consider the arithmetic operations, i.e., add and multiply. (The negation operator is calculated using multiplying by -1 and the arithmetic subtract operator can be obtained using add and negation operators.) Then we show how boolean operations should be computed by using arithmetic operations.

Let *u* and *v* be two nodes to be composed, resulting in a new node *q*. Let var(u)=x and var(v)=y denote the decomposing variables associated with the two nodes. For the simplicity of these algorithms, we do not show the edges's weight, but it is straitforward that the G.C.D of the outgoing edges's weight of each node should be sent to the incoming edge of that node.

3.1. Add operation

To add two nodes of *BTD* representing two functions, first the levels of both nodes are considered. The level of a node is only dependent on the decomposition variable of that node. A node is in higher level than the other, if its corresponding decomposing variable has higher order than the other one in the variable order list. For adding two *BTD* nodes, i.e. *u* and *v*, we should check the following cases:

1. If both nodes have the same index (see Figure 4), we add left children of two nodes together to construct the left child of the result and also two of their right children for its right one.
2. If the nodes are indexed by different variables. Let ord(x) > ord(y), then *v* node must be added to the left child of *u* node (see Figure 5).

This procedure will be applied recursively until we reach the terminal nodes. When only one node is terminal, the process is the same, and when both of them are terminals, then the result is a terminal node with the value equal to the sum of two values of two terminals.

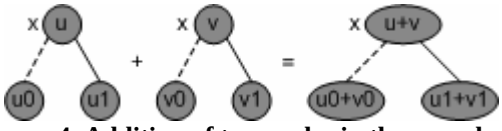


Figure 4. Addition of two nodes in the same level

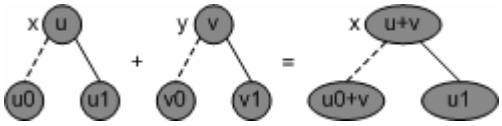


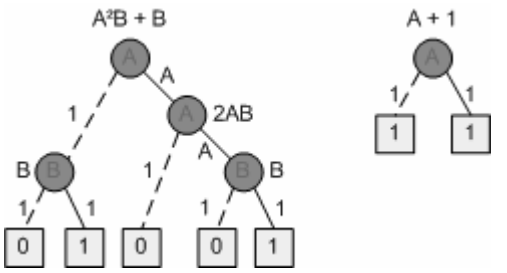
Figure 5. Addition of two nodes in different levels

Figure 6 illustrates this algorithm. Because of binary data structure of the *BTD*, it is easier to implement than *TED*.

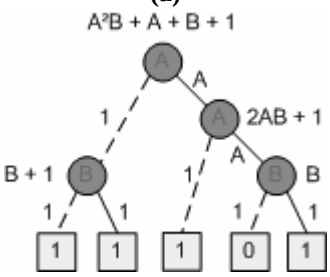
```
function + (btd1, btd2: BTD): BTD;
   result: BTD;
begin
   if btd1 and btd2 are terminal nodes then
      result = btd1.value + btd2.value;
   else
      if btd1 order is greater than btd2 order then
         result.left = btd1.left + btd2;
         result.right = btd1.right;
      else if btd2 order is greater than btd1 order then
         result.left = btd2.left + btd1;
         result.right = btd2.right;
      else // btd1 and btd2 have equal order
         result.left = btd1.left + btd2.left;
         result.right = btd1.right + btd2.right;
      endif
   endif
endfunction
```

Figure 6. Pseudo code of add operation

Example 3: Figure 7(a) illustrates two algebraic expressions A^2B+B and $A+1$. Their addition is performed as Figure 7(b) shows.



(a)



(b)

Figure 7. Example of addition of two BTDs

3.2. Multiply operation

To multiply the two nodes, i.e., *u* and *v*, we consider the following cases:

1. If the levels are the same (see Figure 8), first the left children of two nodes will be multiplied together and the product is assigned in the left children of the result node, then the right children of two nodes are multiplied together and the product is assigned to the right child of the right child (two levels lower, not a typo!) of the result node; and a terminal node 0 is assigned to the left child of right child of the result node. Then a temporary node is employed which is assigned to the sum of the crossed multiplications of the two children nodes. Finally this temporary node is added to the right children of the result node to complete the multiply operation.
2. For cases that the level of two nodes were not the same, let ord(x) > ord(y), then the *v* node is multiplied in both children of the *u* node (see Figure 9).

The algorithm is called recursively until we reach the terminal nodes.

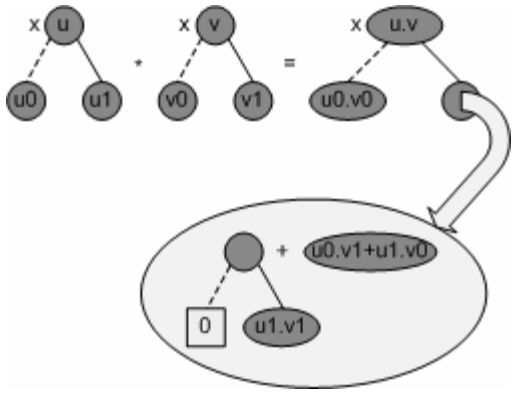


Figure 8. Multiply operation for nodes in the same level

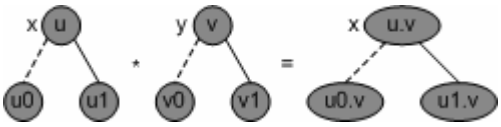


Figure 9. Multiply operation for nodes in different levels

Figure 10 shows this algorithm. As you can see here, the multiply operation of *BTD* is easier than that of *TED* [7] [8] [9], therefore, it is easier to implement and from that, it should be faster than *TED*'s multiply operation.

```
function * (btd1, btd2: BTD): BTD;
    result: BTD;
    t1, t2, t3, t4: BTD;
begin
    if btd1 and btd2 are terminal nodes then
        result = btd1.value * btd2.value;
    else
        if btd1 order is greater than btd2 order then
            result.left = btd1.left * btd2;
            result.right = btd1.right * btd2;
        else if btd2 order is greater than btd1 order then
            result.left = btd2.left * btd1;
            result.right = btd2.right * btd1;
        else // btd1 and btd2 have equal order
            t1 = btd1.left * btd2.right;
            t2 = btd1.right * btd2.left;
            t3 = t1 + t2;

            t4.left = BTD_ZERO;
            t4.right = bt1.right * bt2.right;

            result.left = btd1.left * btd2.left;
            result.right = t3 + t4;
        endif
    endif
endfunction
```

Figure 10. Pseudo code of multiply operation.

Example 4: Figure 11 shows multiplication of two algebraic expressions A^2B+B and $A+1$.

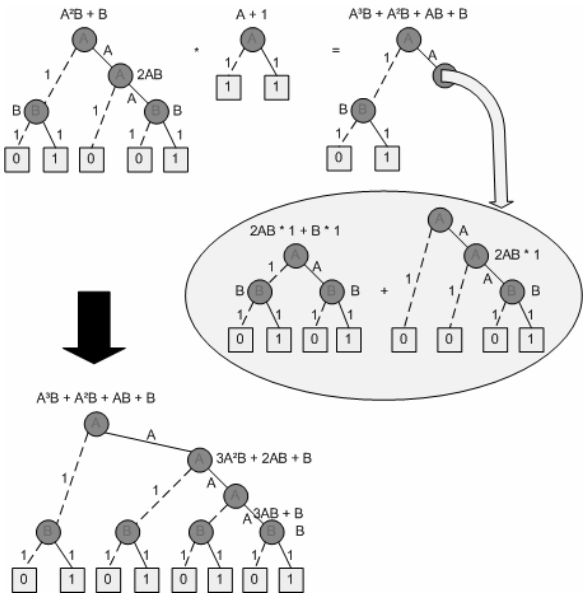


Figure 11. Multiplication of A+1 and $A^2B + B$ by using BTD representation

3.3 Boolean operations

Now we specify how boolean functions are represented by *BTD*. As *BTD* is a kind of Taylor expansion diagrams, the boolean operations of *BTD* are the same as those of *TED* [7] [8] [9]. Employing boolean logic in *BTD* is applicable just with restricting domains and ranges of the functions to {0, 1} and defining boolean operations based on the arithmetic ones. Boolean operations are obtained just by some simple improvements on the arithmetic functions as follows:

NOT (x) = x' = 1-x;
AND (x, y) = x*y;
OR (x, y) = x + y - x*y;
XOR (x, y) = x + y - 2*x*y;

Example 5: Figure 12 shows two boolean circuits. For proving the equivalency of these circuits, first boolean operations are converted to arithmetic operations:

*f = XOR (a, b) = a + b – 2*a*b*
g = AND (OR (a, b), NOT (AND (a, b)))
*= AND (a + b – a*b, 1 – (a*b))*
*= (a + b – a*b) * (1 – a*b)*

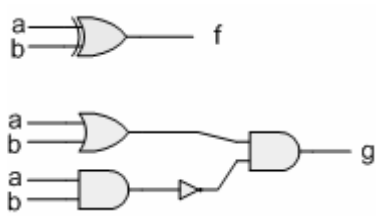


Figure 12. Boolean circuits representing f and g functions

Then the corresponding *BTD* for both designs are built which are the same (see Figure 13).

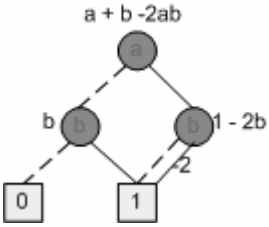


Figure 13. Corresponding *BTD* for f and g functions

4. BTD reduction rules

Again, as *BTD* is a kind of Taylor expansion diagrams, the reduction rules of *BTD* are the same as those of *TED* [7] [8] [9]. To reduce *BTD* as a minimal structure, we can use these reduction rules:

Rule 1: Deleting the garbage nodes. A garbage node is a node which its right child is terminal **0** or its right edge has 0 weight (i.e. only terminal **0** has 0 weight). Therefore, the garbage node is replaced by its left child.

Rule 2: Merging the equivalent (isomorphic) nodes. Two nodes are equivalent; if they have the same index and their children have the same index and are equivalent. To remove redundant nodes, the first one should be deleted. Then all incoming edges to the deleted node should point to the other one.

5. BTD canonicity

Here we will prove that our reduced Binary Taylor Diagram is canonical, i.e., each algebraic function expression has a one-to-one correspondence with a unique Binary Taylor Diagram.

When the variables are ordered and reduction rules remove any redundancy in the graph, and a specific strategy is used to assign weights to the graph edges, canonicity is obvious by inspection.

Proof: The proof of this theorem is conceptually straightforward. We will show that *TED* is canonical and then from that we will prove that *BTD* is also canonical.

In each step of building *TED* of a function named *f(x)*, *f (0)* and all of its derivation (i.e. *f' (0)*, *f'' (0)*, …) should be calculated. For two functions, *g(x)* and *h(x)*, we have:

$$g(0)=h(0), g'(0)=h'(0), g''(0)=h''(0), \cdots \Leftrightarrow f(x)=g(x) \quad \textbf{(4)}$$

This means that two functions are equal if and only if all derivations of these two functions are equal. Form this statement; we can argue that the *TED*s of two functions are the same if and only if these two functions are equal.

Likewise, in each step of building *BTD* of a function named *f(x)*, *f (0)* and its first derivation, *f' (x)*, should be calculated. We do this in each step recursively, till derivation of *f* does not depend on *x*.

In step 1, we have: *f (0)*, *f' (x)*

In step 2, we have: *f' (0)*, *f'' (x)*

In step 3, we have: *f'' (0)*, $f^{(3)}(x)$

…

In step n, we have $f^{(n-1)}(0)$, $f^{(n)}(x)$

And $f^{(n)}(x) = f^{(n)}(0)$ because $f^{(n)}(x)$ does not depend on *x*. Therefore *f (0)* and all of its derivation (the same as *TED*) are used in building *BTD* of *f(x)* and all of them have a specific position in the result *BTD*. Therefore, from equation (4), we can argue that *BTD* is also canonical.■

6. Experimental results

It is obvious that the implementation of the *BTD* is much easier than *TED*, because of its simpler data structure and the simplicity and clarity of the arithmetic operations.

The *BTD* have been implemented. In order to demonstrate the effectiveness of our proposed *BTD*, we have conducted some experiments on some benchmark equations. We have converted these equations to the *BTD* format and have measured the memory usage and *CPU* time of *BTD* corresponding to these equations. Table 1 gives the number of *BTD* nodes and *CPU* time for each equation.

As shown in table 1, *BTD* is feasible in both number of *BTD* nodes and *CPU* time for many typically encountered equations in RTL.

The complexity of both "add" and "multiply" operations for *BTD* is *O (n+m)*, in which *n* and *m* are the number of nodes in two operand *BTD*s. The linear complexity is because in add and multiply operations each node in *BTD* is only traversed once. This can also be proved by our experimental results in Table 1.

Table 1. BTD construction for various equations

	BTD	
Equations	**No. Nodes**	**Time (Seconds)**
$\sum_{i=1}^{100} x_i$	102	0
$\sum_{i=1}^{1000} x_i$	1002	0
$\sum_{i=1}^{10000} x_i$	10002	0.1
$\sum_{i=1}^{100000} x_i$	100002	1.1
$\sum_{i=1}^{1000000} x_i$	1000002	9.3
$\prod_{i=1}^{100} x_i$	102	0
$\prod_{i=1}^{1000} x_i$	1002	0
$\prod_{i=1}^{10000} x_i$	10002	0.1
$\prod_{i=1}^{100000} x_i$	100002	1.2
$\prod_{i=1}^{1000000} x_i$	1000002	11.2
$\prod_{i=1}^{100} \sum_{j=1}^{100} x_{i,j}^2$	19604	0.4
$\prod_{i=1}^{300} \sum_{j=1}^{300} x_{i,j}^2$	178804	3.7
$\prod_{i=1}^{500} \sum_{j=1}^{500} x_{i,j}^2$	498004	10.4
$\prod_{i=1}^{100} \sum_{j=1}^{100} x_{i,j}^3$	29405	0.6
$\prod_{i=1}^{300} \sum_{j=1}^{300} x_{i,j}^3$	268205	5.8
$\prod_{i=1}^{500} \sum_{j=1}^{500} x_{i,j}^3$	747005	18.2
$\prod_{i=1}^{10} \sum_{j=1}^{10} x_{i,j}(i+j)$	812	0
$\prod_{i=1}^{20} \sum_{j=1}^{20} x_{i,j}(i+j)$	7222	0.1
$\prod_{i=1}^{30} \sum_{j=1}^{30} x_{i,j}(i+j)$	25232	0.6
$\prod_{i=1}^{40} \sum_{j=1}^{40} x_{i,j}(i+j)$	60842	1.4

$\prod_{i=1}^{50}\sum_{j=1}^{50} x_{i,j}(i+j)$	120052	2.8
$\prod_{i=1}^{100}\sum_{j=1}^{100} x_{i,j}(i+j)$	980102	23.9
$\prod_{i=1}^{5}(\sum_{j=1}^{5} x_{i,j}{}^{j})^{i}$	678	0
$\prod_{i=1}^{6}(\sum_{j=1}^{6} x_{i,j}{}^{j})^{i}$	4068	0
$\prod_{i=1}^{7}(\sum_{j=1}^{7} x_{i,j}{}^{j})^{i}$	21480	0.3
$\prod_{i=1}^{8}(\sum_{j=1}^{8} x_{i,j}{}^{j})^{i}$	113772	2
$\prod_{i=1}^{9}(\sum_{j=1}^{9} x_{i,j}{}^{j})^{i}$	527097	13.9
$\prod_{i=1}^{10}(\sum_{j=1}^{10} x_{i,j}{}^{j})^{i}$	Unfeasible	Unfeasible

7. Conclusion

BTD is a simple binary tree structure for efficient implementation of *TED* which is minimal and canonical and is suitable for representing both boolean and arithmetic functions in different levels of abstraction. Therefore, *BTD* helps high level verification tools for high level equivalence checking and verification. On the one hand its simplicity and on the other, its generality, make it a powerful system modeler.

From the above discussions it follows that an ordered *BTD* is a canonical method that can be exploited efficiently for *RTL* equivalence checking purposes.

References

[1] Brayant, R. E.; **"Graph-Based Algorithms for Boolean Function Manipulation"** in Computers, IEEE Transactions on , Volume: C-35 , Issue: 8 , Aug. 1985 Pages: 677 – 691

[2] Kebschull, U.; Schubert, E.; Rosenstiel, W.; **"Multilevel logic synthesis based on functional decision diagrams"** in Design Automation, 1992. Proceedings. [3rd] European Conference on , 16-19 March 1992 Pages:43 – 47

[3] Drechsler, R.; Becker, B.; **"Ordered Kronecker functional decision diagrams-a data structure for representation and manipulation of Boolean functions"** in Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on , Volume: 17 , Issue: 10 , Oct. 1998 Pages:965 – 973

[4] Clarke, E.M.; Fujita, M.; Zhao, X.; **"Hybrid decision diagrams. Overcoming the limitations of MTBDDs and BMDs"** in Computer-Aided Design, 1995. ICCAD-95. Digest of Technical Papers., 1995 IEEE/ACM International Conference on , 5-9 Nov. 1995 Pages:159 – 163

[5] Brayant, R. E.; Chen, Y.; **"Verification of arithmetic circuits with binary moment diagrams"** in ACM/IEEE Design Automation Conference, 1995. Proceedings, January 1995

[6] Drechsler, R.; Becker, B.; Ruppertz, S.; **"The K*BMD: A verification data structure"** in Design & Test of Computers, IEEE , Volume: 14 , Issue: 2 , April-June 1997 Pages:51 - 59

[7] Kalla, P.; Ciesielski, M.; Boutillon, E.; Martin, E.; **"High-level design verification using Taylor Expansion Diagrams: first results"** in High-Level Design Validation and Test Workshop, 2002. Seventh IEEE International, 27-29 Oct. 2002 Pages:13 – 17

[8] Ciesielski, M.; Kalla, P.; Zhihong Zeng; Rouzeyre, B.; **"Taylor expansion diagrams: a new representation for RTL verification"** in High-Level Design Validation and Test Workshop, 2001. Proceedings. Sixth IEEE International , 7-9 Nov. 2001 Pages:70 – 75

[9] Ciesielski, M.J.; Kalla, P.; Zhihong Zheng; Rouzeyre, B.; **"Taylor expansion diagrams: a compact, canonical representation with applications to symbolic verification"** in Design, Automation and Test in Europe Conference and Exhibition, 2002. Proceedings, 4-8 March 2002 Pages:285 – 289

[10] Bahar, R.I.; Frohm, E.A.; Gaona, C.M.; Hachtel, G.D.; Macii, E.; Pardo, A.; Somenzi, F.; **"Algebraic decision diagrams and their applications"** in Computer-Aided Design, 1993. ICCAD-93. Digest of Technical Papers., 1993 IEEE/ACM International Conference on , 7-11 Nov. 1993 Pages:188 – 191

[11] Brace, K.S.; Rudell, R.L.; Bryant, R.E.; **"Efficient implementation of a BDD package"** in Design Automation Conference, 1990. Proceedings. 27th ACM/IEEE , 24-28 June 1990 Pages:40 – 45

[12] Parasuram, Y.; Stabler, E.; Shiu-Kai Chin; **"Parallel implementation of BDD algorithms using a distributed shared memory"** in System Sciences, 1994. Vol. I: Architecture, Proceedings of the Twenty-Seventh Hawaii Internation Conference on , Volume: 1 , 4-7 Jan. 1994 Pages:16 – 25

[13] Stornetta, T.; Brewer, F.; **"Implementation of an efficient parallel BDD package"** in Design Automation Conference Proceedings 1996, 33rd , 3-7 June 1996 Pages:641 - 644