Canonical Graph-based Representations for Verification of Logic and Arithmetic Designs

Maciej Ciesielski
Department of Electrical and Computer Engineering
University of Massachusetts, Amherst, MA 01003, USA

Abusaleh M. Jabir
School of Technology
Oxford Brookes University
Oxford OX33 1HX, UK
E-mail: ajabir@brookes.ac.uk

Dhiraj K. Pradhan
Department of Computer Science
University of Bristol
Bristol BS8 1UB, UK
E-mail: pradhan@cs.bris.ac.uk

Index Terms: *Decision Diagrams, Canonical Representaions, Word-level Diagrams, Galois Fields, Verification.*

Abstract

This chapter presents several canonical, graph-based representations used in design and verification of logic and arithmetic circuits on different levels of abstraction. In particular, canonical diagram representations are described for Boolean, word-level, integer, and finite field multiple-output functions. They include Binary Decision Diagrams (BDDs), a number of Word-Level Decision Diagrams (WDDs), Binary Moment Diagrams (BMDs), Taylor Expansion Diagrams (TEDs), and Finite Field Decision Diagrams (FFDDs).
All these diagrams are canonical and minimal with respect to a fixed variable order of their input variables. As such, they support equivalence verification of two combinational designs by checking isomorphism of their graph representations. While BDDs have been used extensively in representing and verifying bit-level designs, such as control and random logic, several WDDs and BMDs find important applications in verifying arithmetic designs with bit-level inputs and integer outputs. TEDs and FFDDs further extend these forms by allowing inputs and outputs to take integer or multiple (finite-field) values. Of particular interest is application of these structures to verification of designs with significant arithmetic component. These computations can often be expressed as multi-variate polynomials and require efficient abstract models for their representation.

1 Introduction

Having matured over the years, formal design verification methods, such as theorem proving, property and model checking, and equivalence checking, have found increasing application in industry. Canonical graph-based representations, such as Binary Decision Diagrams (BDDs) [1], Binary Moment Diagrams (BMDs) [2] and their variants, play an important role in the development of sofware tools for verification. While these techniques are quite mature at the structural level, the high-level verification models are only now being developed. The main difficulty is caused by the fact that such verification must span several levels of design abstraction. Verification of arithmetic designs is particularly difficult because of the disparity in the representations on the different levels and the complexity of logic involved.

This chapter addresses verification based on canonical data structures. It presents several canonical, graph-based representations that are used in formal verification, and in particular in equivalence checking of combinational designs specified at different levels of abstraction. These representations are commonly known under the term "decision diagrams", even though not all of them are actually decision-based forms. They are graph-based structures whose nodes represent the variables and the directed edges represent the result of the decomposition of the function with respect to the individual variables. Particular attention is given to arithmetic and word-level representations.

An important common feature of all these representations is canonicity, which is essential in combinational equivalence checking. A form is canonical if representation of a function in that form is unique. Canonical graph-based representations make it possible to check if two combinational functions are equivalent by checking if their graph-based representations are isomorphic. Checking for isomorphism can be done in constant time, once the representation has been constructed, by testing if the two functions share the same root of the diagram.

The canonical diagrams can be fully characterized by the following basic properties, described in details in this chapter:

1. **Decomposition principle**, which defines the type of function that can be modeled by the diagram and the underlying decomposition method. They include binary decomposition for Boolean functions; some form of multi-valued decomposition for integer-valued functions; and moment decomposition or other non-binary expansions for arithmetic functions.

2. **Simplification rules** that make the diagram minimal and irredundant, hence canonical. Different rules apply to different types of diagrams.

3. **Composition algorithms**, which, given graph-based representations for functions F and G, specify how to construct a similar representation for function $F < op > G$, where $< op >$ represents an arbitrary operation defined for the given application domain (Boolean, arithmetic, finite field, etc). The composition algorithms, commonly known as APPLY algorithms, recursively apply the given operation $< op >$ to the decomposed functions, depending on the type of functions and operations allowed.

One of the most known and commonly used canonical diagram representation is Binary Decision Diagram (BDD) [1]. BDDs are based on the well known Shannon (or more accurately, Boole) function expansion, which decomposes the function into two co-factors, $f(x=0)$ and $f(x=1)$. Each subgraph resulting from such a decomposition can be viewed as as a decision ($x=0$ or $x=1$) taken at a decomposing variable, justifying the name *decision diagram.* BDDs have been developed for Boolean functions and logic circuits represented at the bit level and used extensively in representing and verifying bit-level designs, such as control and random logic. Thanks to their compact, canonical form they truly revolutionized the field of combinational verification and logic synthesis and found applications to many other fields, such as satisfiability, testing and synthesis. However, because of their exponential worst-case size complexity, they have had limited success in modeling and verifying RTL designs with significant arithmetic component, especially with multipliers.

Another canonical form described in this chapter is a Binary Moment Diagram (BMD) [2], developed specifically for arithmetic functions. BMDs are based on a moment decomposition principle, which treats an arithmetic function as a linear function with Boolean inputs and integer (or real-valued) outputs. The two sub-functions resulting from the decomposition represent the two moments (constant and linear) of the function, rather than a "decision". For this reason BMDs do not technically belong to a category of decision diagrams but form a class of their own. BMDs find important applications in verifying arithmetic designs with bit-level inputs and integer outputs.

Two newer types of diagrams, called Taylor Expansion Diagram (TED) and Finite Field Decision Diagram (FFDD), have been recently introduced to address the need for a more abstract design representation, with inputs and outputs allowed to take either integer or discrete (finite-field) values. Both of these diagrams can be thought of as extensions of BDDs and BMDs, with inputs and outputs represented as symbolic variables. The two diagrams differ in arithmetic representation of the data (infinite precision integer vs finite field arithmetic) and the type of decomposition used (Taylor expansion vs multi-valued GF decomposition).

Taylor Expansion Diagram (TED) [3] is based on Taylor expansion of polynomial representation of the computation expressed in the design. Both inputs and outputs are treated as infinite precision integers (or real numbers) and are represented by symbolic variables. The power of abstraction, combined with canonicity and compactness makes the TED particularly attractive for verification of designs specified at the behavioral and algorithmic levels, such as datapaths and signal processing systems. Computations performed by those designs can often be expressed as polynomials and be efficiently represented with TEDs, with memory requirements several orders of magnitude smaller than those of other known representations. TEDs can also serve as a vehicle to transform the initial functional representation of the design into a structural representation in form of a data flow graph (DFG); as such they are applicable to behavioral synthesis, or, more specifically, to behavioral transformations which can also be used in verification.

Finite Field Decision Diagram (FFDD) [4] is an extension of multiple-terminal decision diagrams, but with inputs and outputs represented in finite field (also called Galois Field, GF) arithmetic rather than in the integer domain. Finite field representation has numerous applications in cryptography, error control systems, fault tolerant designs, and digital signal processing. FFDD representation allows the simulation and verification of such systems to be performed more efficiently on a higher level of abstraction. The verification can be performed either at the bit or the word-level; it is not restricted to word boundaries and can be used to model and verify any combination of output bits.

2 Decision Diagrams

Binary Decision Diagrams (BDD) have emerged as representation of choice for many applications, ranging from representation of Boolean function, to verification and satisfiability, to logic synthesis. Even though BDDs (albeit under different name) have been known since late 1950s it was the seminal work of Bryant [1] that brought to light their importance as canonical representations for Boolean logic. This section briefly reviews basic theory and algorithms for BDDs, taken from multiple sources [1,5–7].

2.1 Binary Decision Diagrams (BDD)

A binary decision diagram is a graph-based data structure, which represents a set of binary-valued decisions, culminating at the overall decision that can be either true or false. Specifically, BDD is a directed acyclic graph (DAG) whose nodes represent the decisions, and edges represent the decision types (true or false). The final decision evaluated at the root represents the overall function encoded by the BDD. Ordered and reduced BDD is irredundant and canonical, i.e., a representation of a function in that form is unique. Formally, BDD is defined as follows:

Definition 2.1 *A Binary Decision Diagram (BDD) is a rooted directed acyclic graph $G(V,E)$ with a set of nodes V and set of edges E. The vertex set V contains two types of vertices:*

- *Two terminal nodes (leaves), corresponding to constant 0 and 1.*
- *A set of variable nodes $\{u\}$, each associated with a Boolean variable $v = var(u)$. Each node has exactly two out-going edges, pointing to two children functions, $low(u)$ and $high(u)$. (In the figures, the two children edges are represented as dotted and solid lines, respectively.)*

The function of node $u \in V$, associated with variable $v = var(u)$ is given by $f^u = \bar{v} \cdot low(u) + v \cdot high(u)$, where $low(u)$ and $high(u)$ are the functions of the low and high children of u, respectively. In particular, the function evaluated at the root represents the logic function encoded in the BDD.

The Decomposition Principle. The above definition basically states that BDD is based on a Shannon (Boole) expansion of function f, applied recursively to its variables. That is,

$$f^u = \bar{v} \cdot f_{\bar{v}} + v \cdot f_v \qquad (1)$$

where $f_{\bar{v}} = low(u)$ and $f_v = high(u)$ are the negative and positive cofactors of f with respect to the decomposing variable v.

Definition 2.2 *A BDD is ordered (denoted OBDD) if on all paths from the root to its terminal nodes, the variables appear in the same linear order: $x_1 < x_2 < ... < x_n$. Furthermore, OBDD is reduced (denoted ROBDD) if it satisfies two properties:*

1. *(Irredundancy) No variable node u has identical low and high children, i.e., $low(u) \neq high(u)$.*

2. *(Uniqueness) No two distinct nodes u and v have the same variable name and the same low and high children That is,* $var(u) = var(v),\ low(u) = low(v), high(u) = high(v) \Rightarrow u = v$

The above definition provide the *reduction rules* for BDDs: rule 1 removes redundant nodes with same low and high children; rule 2 merges isomorphic subgraphs. The resulting ROBDDs is an irredundant representation, i.e., no two nodes of the ROBDD represent the same Boolean function. Two ROBDDs are *isomorphic* if there is a one-to-one mapping between the vertex sets that preserves adjacency, indices and leaf values. Thus two isomorphic ROBDDs represent the same function. Conversely, two Boolean expressions that represent the same logic function have isomorphic ROBDDs for a given ordering of variables. In this sense ROBDDs are a canonical representation.

The following lemma, due to Bryant, states the canonicity of ROBDDs [1].

Lemma 1 *For any Boolean function f there is exactly one ROBDD with root node u and variable order* $x_1 < x_2 < ... < x_n$ *such that* $f^u = f(x_1, x_2, ..., x_n)$.

BDD Construction. An algorithm has been proposed by Bryant to reduce OBDD. The resulting diagram, ROBDD, is irredundant, minimal and canonical. The algorithm visits the OBDD bottom up, from the leaf nodes to the root, and labels each vertex $v \in V$ with an identifier $id(v)$. The reduction rules are then applied to remove redundant nodes and merge isomorphic subgraphs. As a result, an ROBDD is identified by a subset of vertices with different identifiers.

The algorithm is illustrated in Figure 1, taken from [5] for function $f = (a+b)c$. OBDD is constructed from the original expression, as shown in Figure 1(a). Then, the nodes of the OBDD are labeled with identifiers, as a function of the variable name and their children. First, the leaf nodes (0 and 1) are labeled with identifiers $id = 1$ and $id = 2$, respectively. Then the vertices v_4, v_5 on the bottom most level, corresponding to variable c, are labeled with their identifiers. In this case both nodes are assigned the same identifier, $id = 3$, since they correspond to the same variable and have children with same identifiers. They are replaced by a single node, v_4 (visited first), added to the ROBDD. Next, the algorithm visits vertices v_2, v_3, associated with variable b. Vertex v_2 is assigned identifier $id = 4$ and is added to the ROBDD. The left (low) and right (high) children of node v_3 have the same identifier, so v_3 inherits their identifier and is discarded as redundant. Finally, the root v_1, associated with variable a is visited and assigned the identifier $id = 5$. It is added to the ROBDD as a unique node with this identifier. The resulting ROBDD is shown in Figure 1(c).

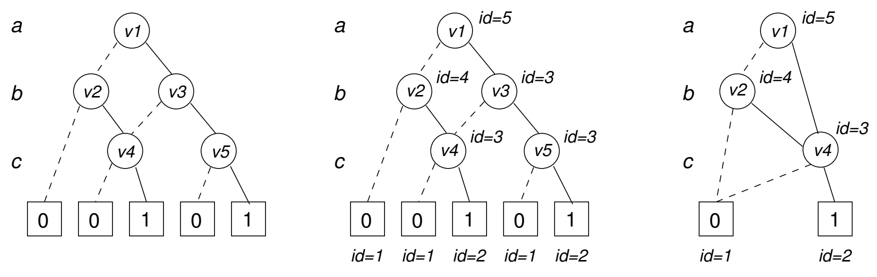


Figure 1: Construction of a ROBDD for $f = (a+b)c$: a) OBDD for the variable order a, b, c; b) OBDD with unique identifiers; c) ROBDD for variable order a, b, c.

In practice, ROBDDs are built directly from a Boolean formulae, avoiding the reduction step and possible memory overflow problems. This approach is based on applying the Shannon decomposition, $f = \bar{v} \cdot f_{\bar{v}} + v \cdot f_v$, iteratively to the variables of the formulae in the predetermined order. Canonicity and minimality of such constructed ROBDD are accomplished by using a hash table, called the *unique table*, which contains a key for each vertex of an ROBDD and which uniquely identifies the function associated with that node. The key is a triple, composed of the variable name and the identifiers of the low and high children. The unique table is constructed bottom up. When a new node is considered for addition to the ROBDD, a lookup in the table determines if another vertex in the table already implements the same functionality by comparing the keys. If this is the case, the pointer of the new node is set to the one existing in the table; otherwise a new entry is made in the table for the new node. This way, no redundant nodes are added to the table and the table represents an ROBDD. The run-time complexity of this and other ROBDD construction algorithms is $O(2^n)$, where n is the number of variables. Similarly, the size of the ROBDD is, in the worst case, exponential. The details of the construction of an ROBDD can be found in [1,5].

BDDs provide a compact representation of Boolean logic. Each path of the BDD from root to terminal node 1 represents a product term (on-set cube) of the function encoded in the BDD. It is computed as a product of variables, along the path, at their respective polarity. For example, for a BDD in Figure 1, a path $\{v_1, v_2, v_4, 1\}$ corresponds to the product term $\bar{a}bc$. Logic function encoded in a BDD is then evaluated as a logical sum (OR) of products terms associated with the on-paths. Similarly, a path from root to terminal node 0 represents a complement of the function. This feature is useful for function complementation, which can be done in constant time by simply exchanging the 0 and 1 terminal nodes.

ROBDDs can naturally represent multiple-output functions by modeling them as ROBDDs with shared subgraphs. In the sequel, we will refer to ROBDD simply as BDD, since some ordering of the variables is always imposed on the BDD, and the OBDD must be reduced in order to be canonical.

BDD Composition - The APPLY **Algorithm.** Another way of constructing a BDD for a given Boolean expression is to compose BDDs of its subexpressions using Boolean connectives, like AND, OR, XOR, etc. The algorithm that performs such a composition is known as the APPLY algorithm.

The basic idea comes, again, from the recursive application of Shannon expansion theorem for arbitrary binary operator $< op >$:

$$f < op > g = \bar{v}(f_{\bar{v}} < op > g_{\bar{v}}) + v(f_v < op > g_v) \tag{2}$$

Starting with the top most variable v in the two functions the formula is applied recursively to all the variables in the order they appear in their respective BDDs (f and g must have compatible ordering for the algorithm to work). If v is the top variable of f and g, then the operator $< op >$ is applied to their respective cofactors. If f does not depend on v, then $f_v = f_{\bar{v}} = f$ and the cofactor is the function itself. The worst case complexity of the APPLY algorithms is $O(n_1 n_2)$, where n_1 and n_2 are the number of variables in the two BDDs.

Using the above algorithm, one can construct a BDD for an arbitrary Boolean network or a gate-level netlist. First, a trivial BDD is build for the variables representing primary inputs, and then BDDs of each expression or logic gate are constructed from the BDDs of their immediate inputs, in the topological order, from primary inputs to primary outputs.

Operations on BDDs can be done in polynomial time of their size (number of nodes). However, the real complexity is hidden in their construction, which is expensive both in time and space. BDDs can be exponential in size and cause memory explosion, especially for designs containing arithmetic functions, such as multipliers (BDDs cannot be built for multipliers larger than 16×16 bits).

Variable Ordering. The size of a BDD strongly depends on the ordering of the variables. The size of the BDD (measured in the number of nodes) is, in the worst case, exponential in the number of variables. ROBDDs representing adder functions are particularly sensitive to the variable order; they can have exponential size in the worst case and a linear size in the best case. There are functions (such as multipliers), whose BDD size is exponential regardless of the ordering. Furthermore, there are functions for which the sum of products (SOP) or product of sums (POS) forms are more compact than the BDDs. Many constraint functions of covering problems fall in this category. Figure 2 shows two BDDs for function $F = x_1x_2 + x_3x_4 + x_5x_6$ constructed with two different ordering of variables, lexicographical, and interleaved. One can see significant difference in BDD size.

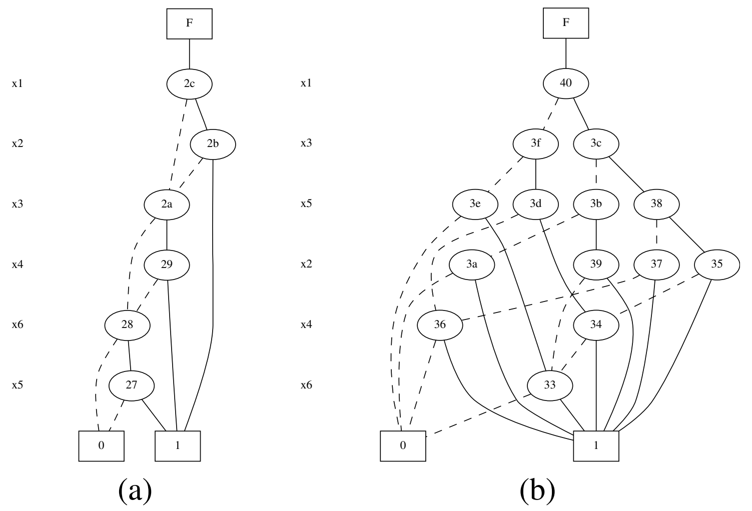


Figure 2: Effect of variable ordering on BDD size for function $F = x_1x_2 + x_3x_4 + x_5x_6$: a) for variable order $x_1, x_2, x_3, x_4, x_5, x_6$ b) for variable order $x_1, x_3, x_5, x_2, x_4, x_6$.

While the variable ordering problem is NP-complete, efficient heuristic variable ordering algorithm exist, based on both static (related to lexicographical) and dynamic ordering (swapping variables on two adjacent levels) [8].

Extensions. Several extensions have been proposed for BDDs. One of them makes a use of *complemented edges* by labeling BDD edges with complement attributes. This feature makes it possible to represent a function and its complement as a single subgraph with two edges coming into the root of the subgraph, one with positive polarity and the other with negative polarity. In general, BDDs with complemented edges result in a smaller BDD size and provide a means to complement a BDD in constant time. To maintain the canonicity, certain restrictions are imposed on the placement of the complemented edges. Namely, only *low* edges, corresponding to negative cofactors, may be assigned complement attributes. Notice that for BDD with complemented edges, only one constant function (1) and hence only one terminal node (leaf 1) is needed, since 0 can be derived from its complement.

Applications and Limitations of BDDs. Owing to their compactness, canonicity and ease of manipulation, BDDs have found numerous applications in design, synthesis, verification and testing of digital designs. In general, BDDs are an efficient data structure for storing and evaluating Boolean function and discrete structures. Large sets of discrete elements can be encoded in binary and compactly represented as characteristic functions in BDDs. BDDs come particularly handy in representing transition relations of product machines for the purpose of sequential equivalence checking using state traversal [7].

In particular, BDDs have found a widespread application in a number of verification problems, including combinational equivalence checking [9], implicit state enumeration and FSM traversal [7, 10], symbolic model checking [11] [12], test vector generation, and many others. The biggest claim to fame comes from their applications to *combinational equivalence checking*. Once two logic functions are represented by their respective ROBDDs (with the same variable order), one can test whether two Boolean functions are equivalent by testing if their ROBDDs are isomorphic. In practice, checking for equivalence between two functions is performed by constructing a single, multi-rooted BDD, rather than checking for graph isomorphism. The two functions are equivalent if they share the same root. This test can be done in constant time once the BDD is built for the two functions.

Logic equivalence can be illustrated with the BDD shown in Figure 1(c). The ROBDD in the figure, constructed for function $f = (a+b)c$, also represents function $g = a \cdot c + b \cdot c$, as well as a number of other equivalent functions, all having the same BDD for a fixed variable order. As mentioned earlier, BDDs can be built for an arbitrary gate-level network or a multiple-output Boolean function. Such created BDD can then be used to check equivalence of the netlist against another netlist, or against the initial Boolean specification of the design.

Another obvious application of BDDs is *satisfiability* (SAT). A Boolean function is satisfiable if there exists an assignment of Boolean values to its variables that makes the function true ($f = 1$). Many verification, synthesis and optimization problems can be reduced to the SAT problem. Being a decision diagrams, BDD can be used to solve the SAT problem in linear time in its size. Once the formula to be satisfied is converted to a BDD, the BDD is traversed to find one or more paths from the root to the terminal node 1. A satisfying solution exists as long as the BDD is not empty. This important feature of decision diagrams finds its application in deterministic *test generation*, used in simulation based verification. A target assignment, not adequately covered by semi-random or directed simulation, is specified and solved using BDD-based SAT.

The BDD-based approach to the SAT problem can be illustrated with the example in Figure 1(c). Two satisfying solution for $f = 1$, corresponding to the paths from the root to node 1, are $\{ac\}$ and $\{\bar{a}bc\}$.

A special case of SAT is related to finding a satisfying assignment for $f = 0$. A notion of *easily invertible* form was introduced by Bryant to denote a representation for which it is always possible to find a zero of the function (solve for $f = 0$) in polynomial time [13]. Clearly, BDDs are easily convertible functions, since one can find a solution to the problem by tracing the path from the root to terminal node 0. Another special case of SAT involves testing for *tautology*, i.e., testing if function is identically equal to 1 (for all assignments of Boolean variables). This can be done in constant time by testing if BDD for the function is reduced to constant 1.

Several efficient implementation of software program supporting BDDs have been developed for a wide set of purposes [14]. One of the most popular packages, available on the world wide web is the CUDD package [15].

In summary, BDDs have been very successful in verifying control-dominated applications and are a part of a number of formal verification systems, such as SMV [12] and VIS [16]. However, as the designs have grown in size and complexity, the size-explosion problems of BDDs have limited their scope. Furthermore, their use in designs containing large arithmetic data-path units have not been very limited due to prohibitive memory requirements, especially for large multipliers.

2.2 Beyond BDDs

In an attempt to obtain a more compact representation for Boolean functions, different flavors of Boolean decomposition have been tried. These diagrams, collectively known as *Decision Diagrams*, are still based on a "point-wise" binary decomposition, but use a different interpretation of the diagram nodes.

One such representation is based on the XOR (exclusive OR) decomposition:

$$f = f_{\bar{x}} \oplus x f_{\Delta x} = f_x \oplus \bar{x} f_{\Delta x} \tag{3}$$

also known as Red-Miller or Davio decomposition. Here, $f_{\Delta x}$ denotes the Boolean difference of function f w.r.t. variable x, i.e., $f_{\Delta x} = f_x \oplus f_{\bar{x}}$, where $\oplus$ represents an XOR operation.

Ordered Functional Decision Diagrams (*OFDDs*) [17] are based on such a decomposition. This representation is analogous to that of OBDDs, except that the two outgoing arcs at each node represent the negative cofactor and the Boolean difference of the function w.r.t. the node variable. Similar to OBDDs, OFDD representation is canonical and many operations can be implemented with algorithms of polynomial complexity. However, several important features differentiate the two representations. First, different reduction rules are applied to make the graph canonical. Second, the evaluations of a function on a OFDDD involves more than tracing a path. In particular for a node variable x, both subgraphs must be evaluated and an XOR computed. Such an evaluation can be performed in linear time in the number of nodes by a postorder traversal of the graph. An interesting feature of the OFDDs is that, for certain classes of functions (in particular, arithmetic functions based on XORs), OFDDs are exponentially more compact than ROBDDs, but the reverse is also true. To obtain the advantages of each representation, Drechsler *et al* proposed a hybrid form, called Ordered Kronecker FDD (OKFDD) [18]. In this representation, each variable can use any of the three decompositions given by equations 1 - 3, potentially leading to a reasonable reduction in the graph size.

Another variant of BDD representation, called Zero-Suppressed BDDs (ZBDDs), was developed by Minato for solving combinatorial problems [19]. ZDDS are particularly suitable for applications involving sparse sets of bit vectors. It can be shown that ZBDDs reduce the size of the representation of a set of n-bit vectors over OBDDs by at most a factor of n. In practice, the reduction is large enough to have a significant impact.

Numerous attempts have been made to extend the capabilities of BDDs to target arithmetic circuits and designs with word-level specifications. This requires extending the concept of Boolean function representation to integer and real-valued functions over Boolean variables. The resulting graph-based representations are commonly known as *Word Level Decision Diagrams* (WLDDs) [20, 21].

One straightforward way to represent numeric-valued functions is to use branching structure of a BDD, but to allow arbitrary values on the terminal nodes. Such a representation is referred to as Multi-terminal BDD (MTBDD) [22] or Algebraic Decision Diagram (ADD) [23]. Evaluating an MTBDD or ADD for a given variable assignment is similar to that of BDD. However, MTBDDs are inefficient in representing functions yielding values over a large range, as this requires a large number of terminal nodes and results in a large number of paths (MTBDDs tend to be trees rather than graphs).

For such applications, alternative representations have been proposed, such as edge-valued BDDs (EVBDDs) [24]. These forms incorporate numeric weights on the edges in order to allow greater sharing of subgraphs and to reduce the size of the overall representation. Evaluating a function represented by an EVBDD involves adding the products of variable values along the path, weighted by the corresponding edge weights. This representation grows linearly in the number of bits, a major improvement over MTBDDs. However, the overhead for storing and normalizing the edge weights to make the representation canonical makes it them less efficient. There are classes of functions, such as arithmetic functions, for which EVBDD has unacceptable size complexity. In particular, the EVBDD representation for integer multipliers, $F = X \cdot Y$, grows exponentially with the number of bits of its operands. A good review of WLDDs can be found in [20, 21].

In the next section, another type of word-level diagram is described, based on a different, non-pointwise decomposition principle.

3 Binary Moment Diagrams (BMD)

An alternative approach to representing numeric functions, especially those encountered in arithmetic circuits, involves changing the function decomposition with respect to its variables.

The Decomposition Principle. Binary Moment Diagrams (BMD), introduced by Bryant [13], use a modified Shannon's expansion, in which a Boolean variable is treated as a binary, (0,1) integer variable. The complement of x is modeled as $\bar{x} = 1 - x$, and the terms of the expansion are regrouped around variable x resulting in the following formula:

$$\begin{aligned} f(x) &= (1-x) \cdot f_{\bar{x}} \, + \, x \cdot f_x \\ &= f_{\bar{x}} \, + \, x \cdot (f_x - f_{\bar{x}}) \\ &= f_{\bar{x}} \, + \, x \cdot f_{\Delta x} \end{aligned} \tag{4}$$

where "$\cdot$", "+" and "-" denote multiplication, addition and subtraction, respectively. The above decomposition is termed as *moment* decomposition; $f_{\bar{x}}$ is the *constant moment*, and $(f_{\Delta x} = f_x - f_{\bar{x}})$

is the *linear moment*. In this form, f can be viewed as a *linear function* in x, with $f_{\bar{x}}$ as the constant term, and $(f_{\Delta x}$ as the linear coefficient of f (the partial derivative of f with respect to x). This expansion still relies on the assumption that variable x is Boolean, i.e., evaluates to either 0 or 1. However, it departs from a point-wise, decision-based decomposition and performs the decomposition of a linear function based on its first two moments.

Each node of a BMD describes a function in terms of its moment decomposition with respect to the variable labeling the node, as shown in Figure 3(a). The two outgoing arcs from each node denote the constant moment (shown as dashed line) and the first moment (solid line) of the function w.r.t. the decomposing variable. Part (b) of the figure shows the BMD representation of the unsigned integer $X = 4x_2 + 2x_1 + x_0$ encoded with $n = 3$ bits. The constants in the terminal nodes of the BMD can be moved from to their edges, and represented as edge-weights, as shown in Figure 3(c). The resulting diagram is termed Multiplicative Binary Moment Diagram, or *BMD. The term "multiplicative" derives from the fact that, when evaluating a function along a path from root to one of its terminal nodes, the weights combine multiplicatively along the path.

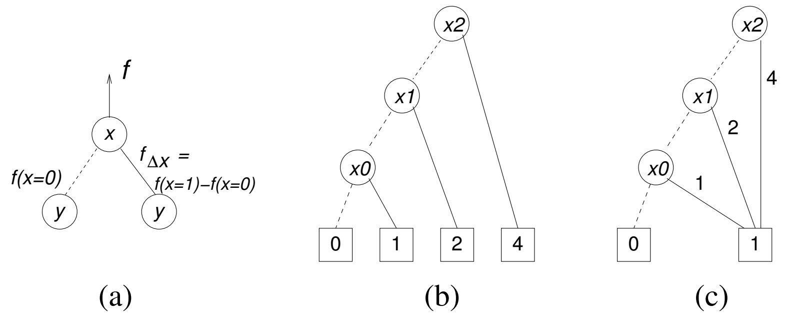


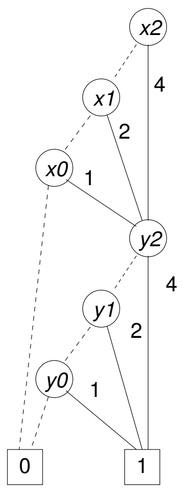
Figure 3: Binary Moment Diagrams: (a) The moment decomposition principle; (b) BMD for binary encoded integer $X = 4x_2 + 2x_1 + x_0$; (c) *BMD for X.

Similarly to BDDs, function encoded in a *BMD is evaluated by adding the terms encoded in the paths. However, two major features differentiate (*)BMDs from decision diagrams discussed earlier: 1) *BMDs are *not* decision diagrams, since they are based on the moment decomposition, rather than a point-wise Shannon expansion. 2) *BMDs are *multiplicative* diagrams, in the sense that each path from root to terminal node is a product of the variables labeling the nodes and the edge weights along the path.

Figure 4 shows *BMDs for addition and multiplication expressed at word levels. Note that the size of *BMDs for these operations grows linearly with the word size n.

Reduction Rules Each node in the *BMD is represented as a triple $< v, low(v), high(v) >$, with two weights associated with the constant and linear moments, $w_0(v), w_1(v)$. It is assumed that the set of variables is totally ordered, like in a BDD. To maintain the canonical form certain reduction rules must be imposed on the *BMD during node creation and weight manipulation (normalization). In principle, these rules are similar to those in BDDs, but must follow rules of regular algebra over $(+,\cdot)$ rather than Boolean algebra $(\vee,\wedge)$.

1. (Irredundancy) When a linear moment at node v is 0, the function at the node evaluates to its constant moment, i.e., does not depend on v. In this case node v is redundant and is removed. (Note that this rules differs from the redundancy reduction rule for BDD).



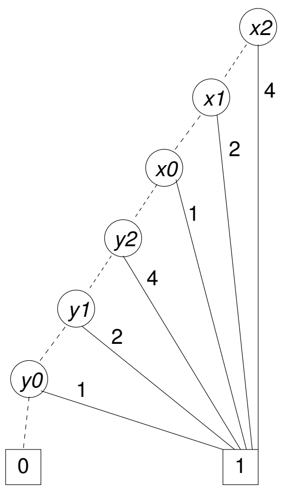


Figure 4: *BMD representations for word-level operations: (a) Sum $X+Y$; (b) Product $X \cdot Y$.

2. (Uniqueness) This rule is similar to that of BDD: any two nodes indexed by the same variable and have same two moments represent the same function and are merged in the BMD into a single node. This rule, however, is applied after the normalization, described next.

Normalization: Several rules for manipulating edge weights are imposed on the graph to make the graph canonical. For non-zero value of the linear moment at node v, the weights of its two edges are normalized by factoring out the greatest common divisor (gcd) of the argument weights $w = gcd(w_0(v), w_1(v))$, which is then pushed to the root edge of node v. By convention, the sign of the extracted weight must match the sign of the constant moment; this way gcd always returns non-negative value. Normalization is performed bottom up, from the leaf nodes to the root. Each normalized node is stored in the hash table, where each entry is indexed by a key composed of the variable and the two moments. Duplicate entries are automatically removed, resulting in an irredundant, minimal and canonical representation.

As with BDDs, the *BMD representation of a function depends on the variable ordering, but *BMDs are much less sensitive to variable ordering than BDDs.

The APPLY **Algorithms.** Similarly to BDDs, *BMDs are constructed by starting with base functions, corresponding to constants and single variables, and then building more complex functions according to some operation. Algorithms similar to the APPLY algorithm for BDDs have been proposed. However, while there is a single APPLY algorithm for BDDs for an arbitrary Boolean operator, *BMDs require algorithms tailored specifically for the individual operations, such as ADD, SUB and MULT [13]. In general,

$$f < op > g = (f < op > g)_{\bar{x}} + x(f < op > g)_{\Delta x} \tag{5}$$

where

$$(f < op > b)_{\bar{x}} = (f_{\bar{x}} < op > g_{\bar{x}}) \tag{6}$$

and

$$(f < op > g)_{\Delta x} = (f < op > g)_x - (f < op > g)_{\bar{x}} \tag{7}$$

$$= (f_x < op > g_x) - (f_{\bar{x}} - g_{\bar{x}})$$
$$= ((f_x + f_{\Delta x}) < op > (g_x + g_{\Delta x}) - (f_{\bar{x}} - g_{\bar{x}})$$

However, in case of the multiply operation, special attention must be paid because of the introduction of the term containing x^2.

$$f \cdot g = (f_{\bar{x}} + x \cdot f_{\Delta x}) \cdot (g_{\bar{x}} + x \cdot g_{\Delta x}) = f_{\bar{x}} \cdot g_{\bar{x}} + x \cdot (f_{\bar{x}} \cdot g_{\Delta x} + f_{\Delta x} \cdot g_{\bar{x}}) + x^2 \cdot f_{\Delta x} \cdot g_{\Delta x} \tag{8}$$

The multiply operation must be linearized by replacing x^2 with x, since x is a Boolean variable. This gives the following result for multiplication

$$f \cdot g = f_{\bar{x}} \cdot g_{\bar{x}} + x \cdot (f_{\bar{x}} \cdot g_{\Delta x} + f_{\Delta x} \cdot g_{\bar{x}} + f_{\Delta x} \cdot g_{\Delta x}) \tag{9}$$

The APPLY algorithms proceed by traversing the argument graphs and recursively apply the operation to the subgraphs. To reduce the number of recursive calls, a hash table is maintained keyed by the arguments of the previous calls, allowing the program to reuse previous computations.

Unlike operations on BDDs, which have run-time complexities polynomial in the number of variables, most operations on *BMDs potentially have exponential complexity. However, as demonstrated by Bryant, these exponential cased do not arise in practical applications [13]. Furthermore, the size of the arguments is significantly smaller in word-level applications than in bit-level applications, resulting in reasonable run-times.

For word-level expressions $(X+Y)$ and $(X*Y)$, where X and Y are n-bit vectors, *BMD representation is linear in the number of bits n. Also, function c^X, where c is a constant, have linear size representation in *BMD. However, the size of *BMD for X^k is $O(n^k)$. Thus, for high-degree polynomials defined over words with large bit-width, as commonly encountered in many DSP applications, filters, etc., *BMD remains an expensive representation.

Boolean Logic. *BMD can be adapted to also represent Boolean logic, which is important for designs with Boolean connectives. The following equations are used to model Boolean logic.

$$NOT: \quad \bar{x} = \quad (1-x) \tag{10}$$
$$AND: \quad x \wedge y = \quad x \cdot y \tag{11}$$
$$OR: \quad x \vee y = \quad x + y - x \cdot y \tag{12}$$
$$XOR: \quad x \oplus y = \quad x + y - 2x \cdot y \tag{13}$$

Figure 5 shows BMD representations for these basic Boolean operators [2]. . In the diagrams, x and y are Boolean variables represented by binary variables, and $+$ and $\cdot$ represent algebraic operators of ADD and MULT, respectively. The resulting functions are 0,1 integer functions.

*BMDs provide a concise representation of functions defined over bit vectors, or words of data, having a numeric representation. In particular they can efficiently encode integer-valued functions defined over binary-encoded words, $X = \sum_i 2^i x_i$, where each x_i = 0 or 1. Figure 6 shows examples of *BMD representation for signed integers several sign schemes (signed magnitude, ones complement, and twos complement). All commonly used encodings can be similarly represented.

Applications to Word-level Verification. (*)BMDs have been successfully used in formal verification of arithmetic circuits. Figure 7 illustrates an approach to arithmetic circuit verification

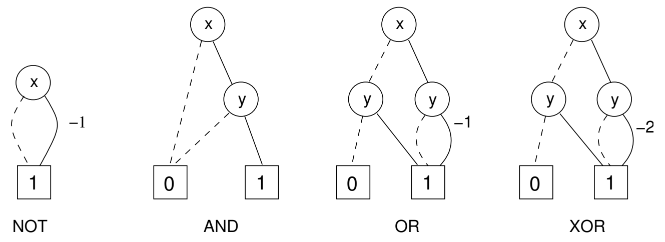


Figure 5: *BMD representation for Boolean operators: a) NOT: $\bar{x} = (1-x)$; b) AND: $x \wedge y = x \cdot y$; c) OR: $x \vee y = x + y - xy$; d) XOR: $x \oplus y = x + y - 2xy$.

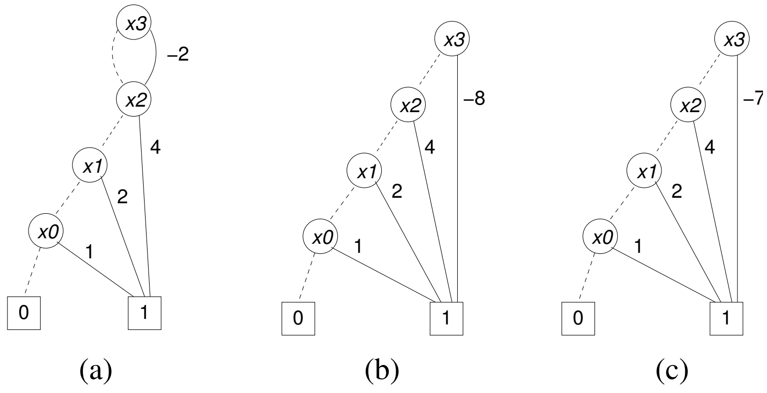


Figure 6: *BMD representations for word-level operations: a) Sign magnitude; b) Two's complement; c) One's complement.

proposed in [13,25]. The goal is to prove a correspondence between a logic circuit, represented by a vector of Boolean functions f, and the design specification, represented by a word-level function F. The inputs to the Boolean circuit f are vectors of Boolean signals, $x_1, x_2, \ldots, x_k$; the inputs to the specification function F are word-level signals (symbolic variables) $X_1, X_2, \ldots, X_k$. In order to compare the two designs, each of the Boolean vectors x_i is transformed into a word-level signal X_i using an encoding function $Enc_i(x_i)$, and connected to the appropriate input of F. An encoding function simply provides the interpretation of the bit vectors. An example of such an encoding function (in this case, unsigned integer) is shown in Figure 3. Similarly, the output of the logic circuit f is transformed to a word-level function using an encoding function Enc_0. The general task of verification is then to prove the equivalence between the circuit output, interpreted as a word, and the output of the word-level specification.

$$Enc_0(f(x_1, \ldots, x_k)) = F(Enc_1(x_1), \ldots, Enc_k(x_k)) \tag{14}$$

*BMDs can provide suitable data structure for this form of verification. TEDs, described in the next section, can be used for for the final comparison at the word-level.

A serious limitation of (*)BMDs is that they cannot be used for solving SAT problems. This is because (*)BMDs are multiplicative diagrams, i.e., the weights combine multiplicatively along the path from terminal node to root. Solving the integer-valued SAT problem in this structure is equivalent to solve the integer factorization problem. They are also not "easily invertible", as defined earlier in the context of the decision diagrams.

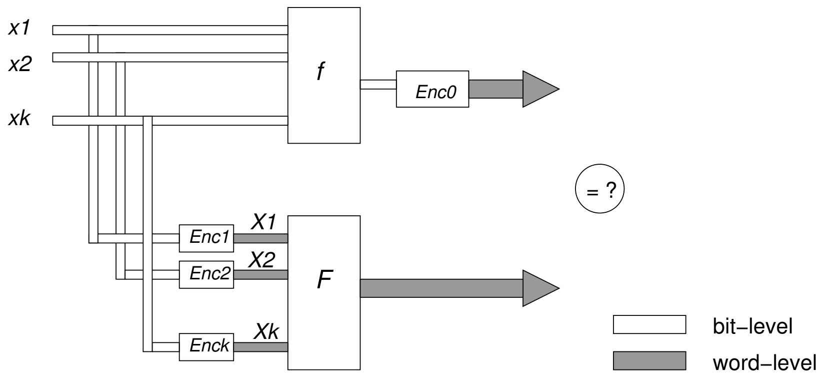


Figure 7: General verification problem: prove correspondence between a word-level specification and bit-level implementation.

Several variants of *BMD representation have been proposed in literature. Chen *et al* introduced Multiplicative Power Hybrid Decision Diagrams (PHDDs) that allows it to handle floating point arithmetic [26]. This is the only known form that supports floating point operation in a graph without introducing rational numbers, by representing the mantissa and exponent as connected subgraphs. However, the size of the graph, even for the adder function, grows exponentially with the size of the exponent size.

Drechsler *et al* extended *BMDs to a form called K*BMD to make the decomposition more efficient in terms of the graph size. This is done by admitting multiple decomposition types and allowing both additive and multiplicative edge weights [27]. However, a set of restrictions imposed on the edge weights to make it canonical makes such a graph difficult to construct. K*BMD is characterized by linear complexity of the word-level operations for sum, product and c^X, and can represent X^k in $O(n^{k-1})$ nodes. As we will see in the next section, this result can be further improved with TEDs [3], which offer linear size complexity for this and other word-level operations.

4 Taylor Expansion Diagrams (TED)

Before formally introducing TEDs, we briefly review previous work and recent advances in word-level equivalence checking and the supporting symbolic representations.

4.1 Related Work.

In the realm of high-level design verification, the issue of abstraction of symbolic, word-level computations have received a lot of attention. This is visible in theorem-proving techniques, automated decision procedures for Presburger arithmetic [28] [29], techniques using algebraic manipulation [30], symbolic simulation [31], or in the decision procedures that use a combination of theories [32] [33]. *Term rewriting* systems, particularly those used for hardware verification [34–36] also represent computations in high-level symbolic forms. The above representations and verification techniques, however, do not rely on canonical forms. For example, verification techniques using term rewriting are based on rewrite rules that lead to normal forms. Such forms may produce false negatives, which may be difficult to analyze and resolve.

Various forms of *high-level logics* have been used to represent and verify high-level design specifications. Such representations are mostly based on quantifier free fragments of first order logic. The works that deserve particular mention include: the logic of equality with uninterpreted functions (EUF) [37] and with memories (PEUFM) [38] [39], and the logic of counter arithmetic with lambda expressions and uninterpreted functions (CLU) [40]. These logics are often transformed into canonical representations, such as BDDs and BMDs, or into SAT instances or other normal forms [41] [42]. To avoid exponential explosion of BDDs, equivalence verification is generally performed by transforming high-level logic description of the design into propositional logic formulas [40] [33] [38] and employing satisfiability tools [43] [44] for testing the validity of the formulas. While these techniques have been successful in the verification of control logic and pipelined microprocessors, they have found limited application in the verification of large datapath designs.

Word-Level ATPG techniques [45] [46] [47] [48] [49] have also been used for RTL and behavioral verification. However, their applications are generally geared toward simulation, functional vector generation or assertion property checking, but not so much toward high-level equivalence verification of arithmetic designs.

Symbolic Algebra Methods. Many computations encountered in behavioral design specifications can be represented in terms of polynomials. This includes digital signal and image processing designs, digital filter designs, and designs that employ complex transformations, such as DCT, DFT, WHT, etc. Polynomial representations of discrete functions have been explored in literature long before the advent of contemporary canonical graph-based representations. Particularly, Taylor's expansion of Boolean functions has been studied in [50] [51]. However, these works mostly targeted classical switching theory problems: logic minimization, functional decomposition, fault detection, etc. The issue of abstraction of bit-vectors and symbolic representation of computations for high-level synthesis and formal verification was not their focus.

Commercial symbolic algebra tools, such as Maple [52], Mathematica [53], and MatLab [54], use advanced symbolic algebra methods to perform efficient manipulation of of mathematical expressions, including fast multiplication, factorization, etc. However, despite the unquestionable effectiveness of these methods for classical mathematical applications, they are less effective in modeling of large scale digital circuits and systems, and in particular in polynomial verification. For example, symbolic algebra tools offered by Mathematica and alike cannot unequivocally determine the *equivalence* of two polynomials. The equivalence is checked by subjecting each polynomial to a series of *expand* operations and comparing the coefficients of the two polynomials ordered lexicographically. As stated in the manual of Mathematica 5, *"it would be quite impossible for Mathematica to match patterns by mathematical, rather than structural, equivalence."* And *"there is no general way to find out whether an arbitrary pair of mathematical expressions are equal"* [53]. Furthermore, Mathematica *"cannot guarantee that any finite sequence of transformations will take any two arbitrarily chosen expressions to a standard form."*

In contrast, the TED data structure described in the sequel provides an important support for equivalence verification by offering a canonical representation for multi-variate polynomials.

Equivalence Checking. Equivalence checking has been researched thoroughly and vast literature exists on the topic, including satisfiability (SAT) approaches [45, 48, 49, 55, 56] verification of arithmetic on bit-level [57–59], symbolic approaches and others [60–66].

Typical approach to equivalence checking (EC) employed by industrial tools, involves identifying structural equivalences or ”similarities” between pairs of points (called *cut points*) in the two designs. The portions of designs identified as having equivalent cut points are removed from the design and the EC verification is repeated on the reduced designs. However, the main difficulty lies in identifying such cut points in designs described in different levels (e.g., RTL and algorithmic). Another challenge in EC verification comes from structural optimizations, employed by behavioral or high-level synthesis (such as factorization, resource sharing, change of order of operators, operator merging, etc.), which reduce the level of similarity between the candidate cut points. The next section provides a motivating example for the development of symbolic equivalence technique based on functional, rather than structural approach and the associated canonical representation.

4.2 Motivation

The following example, Figure 8(a),(b), taken from the Synopsys technical bulletin [67], illustrates the perceived difficulty of functional verification of arithmetic designs in case of a combinational transformation, called resource sharing. Resource sharing transforms the netlist by moving the operators in order to maximize sharing of the resources, in this case the multiplication. Arithmetic Proof Engine (APF) of Synopsys' Formality tool cannot solve this problem using cut-points because internally equivalent points are lost during such a transformation.

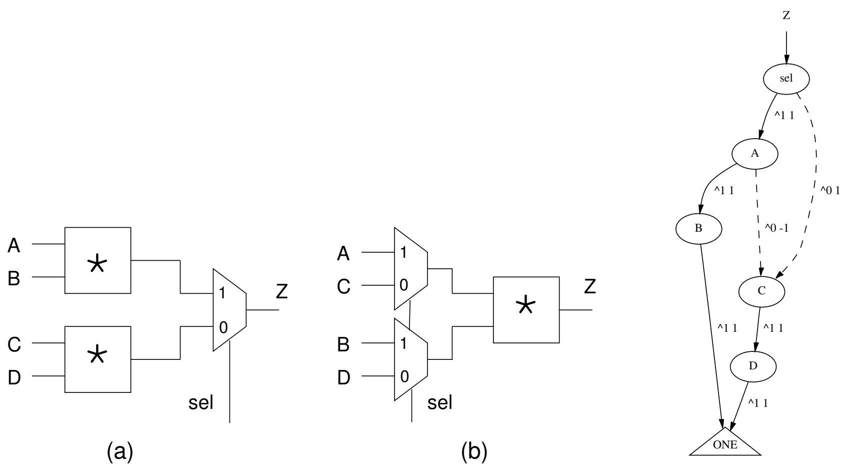


Figure 8: Verification of resource sharing: (a) $z = sel\ ?\ (A \cdot B) : (C \cdot D)$; (b) $z = (sel\ ?\ A : C) \cdot (sel\ ?\ B : D)$; (c) Canonical TED showing functional equivalence of the two structures: $z = A \cdot B \cdot sel + C \cdot D \cdot (1 - sel) = (A \cdot sel + C \cdot (1 - sel)) \cdot (B \cdot sel + D \cdot (1 - sel))$ for $sel = 0, 1$.

This problem can be solved, however, by generating the symbolic expressions for the original and the transformed forms in a canonical form, and proving that they are equivalent. Namely, the function computed by the original design shown in Figure 8(a) can be written as

$$z = A \cdot B \cdot sel + C \cdot D \cdot (1 - sel) \tag{15}$$

while the design in Figure 8(b) can be expressed as

$$(A \cdot sel + C \cdot (1 - sel)) \cdot (B \cdot sel + D \cdot (1 - sel)) \tag{16}$$

Since sel is a binary variable, $sel^2 = sel$ and $sel \cdot (1 - sel) = 0$, and the above expression reduces to the expression (16). These expressions can be captured by a canonical data structure with symbolic input variables A, B, sel. Such a diagram is shown in Figure 8(c). The equivalence of the two designs can be verified by testing if the diagrams corresponding to the two designs are isomorphic, which is the case here (only one graph is shown). This is the main idea behind Taylor Expansion Diagrams, described next. Note that, unlike BMDs, this diagram represents the designs with arbitrary bit-width; that is the designs can verified for equivalence regardless of their word sizes, assuming infinite precision arithmetic.

4.3 The Taylor Series Expansion

A known limitation of all decision and moment diagram representations is that *word-level* computations, such as $A + B$, require the function to be decomposed with respect to *bit-level* variables $A[k], B[k]$. Such an expansion creates a large number of variables in the respective diagram framework and requires excessive memory and time to operate upon them. In order to efficiently represent and process the HDL description of a large design, it is desirable to treat the word-level variables as *algebraic symbols*, expanding them into their bit-level components only when necessary.

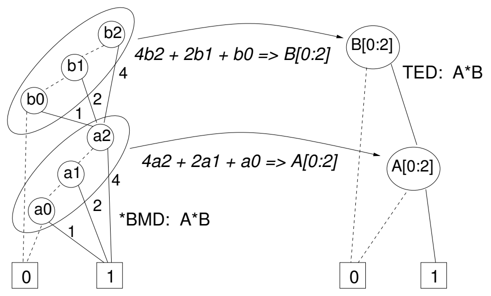


Figure 9: Abstraction of bit-level variables into algebraic symbols for $F = A \cdot B$.

Consider the *BMD for $A \cdot B$, shown in Fig. 9 (a), which depicts the decomposition with respect to the bits of A and B. It would be desirable to group the nodes corresponding to the individual bits of these variables to *abstract* the integer variables they represent, and use the abstracted variables directly in the design. Fig. 9 depicts the idea of such a *symbolic abstraction* of variables from their bit-level components.

In order to achieve the type of abstracted representation depicted above, one can rewrite the moment decomposition $f = f_{\overline{x}} + x \cdot (f_x - f_{\overline{x}})$ as $f = f(x = 0) + x \cdot \frac{\partial(f)}{\partial x}$. This equation resembles a truncated *Taylor series expansion* of the linear function f with respect to x. By allowing x to take integer values, the binary moment decomposition can be generalized to a Taylor's series expansion, where integer variables do not need to be expanded into bits.

In this approach, an algebraic, multi-variate expression, $f(x,y,...)$, can be viewed as continuous, differentiable function oever a real domain. It can be decomposed using the Taylor series expansion with respect to variable x as follows [68]:

$$f(x) \;=\; \sum_{k=0}^{\infty} \frac{1}{k!}(x-x_0)^k f^k(x_0) \;=\; f(0)+xf'(0)+\frac{1}{2}x^2 f''(0)+.... \tag{17}$$

where $f'(x_0)$, $f''(x_0)$, etc., are first, second, and higher order derivatives of f with respect to x, evaluated at $x_0 = 0$. The derivatives of f evaluated at $x = 0$ are independent of variable x, and can be further decomposed w.r.t. the remaining variables, one variable at a time. The resulting recursive decomposition can be represented by a decomposition diagram, called the *Taylor Expansion Diagram*, or TED.

The Taylor series expansion can be used to represent computations over integer and Boolean variables, commonly encountered in HDL descriptions. Arithmetic functions and dataflow portions of those designs, can be expressed as multi-variate polynomials of finite degree, for which Taylor series is finite.

Definition 4.1 *The* **Taylor Expansion Diagram***, or* **TED***, is a directed acyclic graph* (Φ,V,E,T)*, representing a multi-variate polynomial expression* Φ*. V is the set of nodes, E is the set of directed edges, and T is the set of terminal nodes in the graph. Every node* $v \in V$ *has an index* $var(v)$ *which identifies the decomposing variable. The function at node v is determined by the Taylor series expansion at* $x = var(v) = 0$*, according to equation 17. The number of edges emanating from node v is equal to the number of nonempty derivatives of f (including* $f(0)$*) w.r.t. variable* $var(v)$*. Each edge points to a subgraph whose function evaluates to the respective derivative of the function with respect to* $var(v)$*. Each subgraph is recursively defined as TED w.r.t. the remaining variables. Terminal nodes evaluate to constants.*

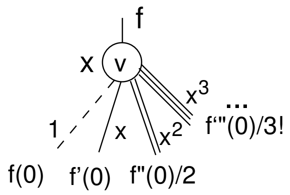


Figure 10: A decomposition node in a TED.

Starting from the *root*, the decomposition is applied recursively to the subsequent children nodes. The internal nodes are in one-to-one correspondence with the successive derivatives of function f w.r.t. variable x evaluated at $x = 0$. Figure 10 depicts one-level decomposition of function f at variable x. The k-th derivative of a function rooted at node v with $var(v) = x$ is referred to as a *k-child* of v; $f(x{=}0)$ is a 0-child, $f'(x{=}0)$ is a 1-child, $\frac{1}{2!}f''(x{=}0)$ is a 2-child, etc. We shall also refer to the corresponding arcs as *0-edge* (dotted), *1-edge* (solid), *2-edge* (double), etc.

Example: Figure 11 shows the construction of a TED for the algebraic expression $F = A^2 + A \cdot (B + 2 \cdot C + 2 \cdot B \cdot C)$. Let the ordering of variables be A, B, C. The decomposition is performed first with respect to variable A. The constant term of the Taylor expansion $F(A = 0) = 2 \cdot B \cdot C$. The linear term of the expansion gives $F'(A = 0) = B + 2 \cdot C$; the quadratic term is $\frac{1}{2} \cdot F''(A = 0) = \frac{1}{2} \cdot 2 = 1$. This decomposition is depicted in Fig. 11 (a). Now the Taylor series expansion is applied recursively to the resulting terms with respect to variable B, as shown in Fig. 11(b), and subsequently with respect to variable C. The resulting diagram is depicted in Fig. 11(c), and its final reduced and normalized version (to be explained in Section 4.4) is shown in Fig. 11(d). The function encoded by the TED can be evaluated by adding all the paths from non-zero terminal nodes to the root, each path being a product of the variables in their respective powers and the edge weights, resulting in $F = A^2 + AB + 2AC + 2BC$.

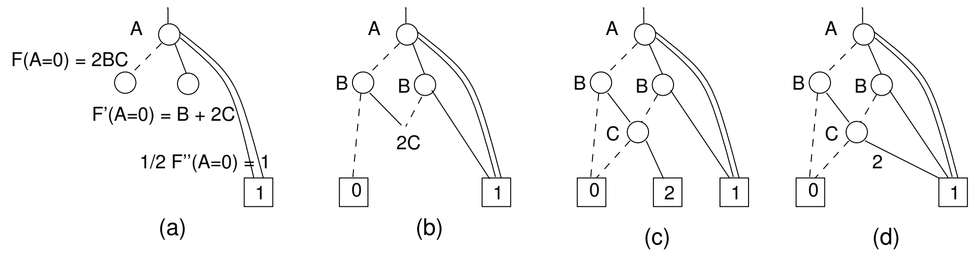


Figure 11: Construction of a TED for $F = F = A^2 + AB + 2AC + 2BC$: (a)-(c) decomposition w.r.t. individual variables; (d) normalized TED

Using the terminology of computer algebra [69], TED employs a *sparse recursive representation*, where a multivariate polynomial $p(x_1, \cdots, x_n)$ is represented as:

$$p(x_1, \cdots, x_n) = \sum_{i=0}^{m} p_i(x_1, \cdots, x_{n-1}) x_n^i \tag{18}$$

The individual polynomials $p_i(x_1, \cdots, x_{n-1})$ can be viewed as "coefficients" of the leading variable x_n at the decomposition level corresponding to x_n. By construction, the sparse form stores only non-zero polynomials as the nodes of the TED.

4.4 Reduction and Normalization

It is possible to further reduce the size of an ordered TED by a process of TED *reduction* and *normalization*. Analogous to BDDs and *BMDs, Taylor Expansion Diagrams can be reduced by removing redundant nodes and merging isomorphic subgraphs. In general, a node is redundant if it can be removed from the graph, and its incoming edges can be redirected to the nodes pointed to by the outgoing edges of the node, without changing the function represented by the diagram.

Definition 4.2 *A TED node is* **redundant** *if all of its non-0 edges are connected to terminal 0.*

If node v contains only a constant term (0-edge), the function computed at that node does not depend on the variable $var(v)$, associated with the node. Morover, if all the edges at node v point

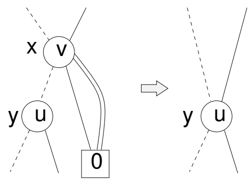


Figure 12: Removal of redundant node with only a constant term edge.

to the terminal node 0, the function computed at the node evaluates to zero. In both cases, the parent of node *v* is reconnected to the 0-child of *v*, as depicted in Fig. 12.

Identification and merging of isomorphic subgraphs in a TED are analogous to that of BDDs and *BMDs. Two TEDs are considered *isomorphic* if they match in both their structure and their attributes; *i.e.* if there is a one-to-one mapping between the vertex sets and the edge sets of the two graphs that preserve vertex adjacency, edge labels and terminal leaf values. By construction, two isomorphic TEDs represent the same function. In order to make the TED canonical, any redundancy in the graph must be eliminated and the graph must be reduced. The reduction process entails merging the isomorphic sub-graphs and removing redundant nodes.

Definition 4.3 *A Taylor expansion diagram is* **reduced** *if it contains no redundant nodes and has no distinct vertices v and v', such that the subgraphs rooted at v and v' are isomorphic. In other words, each node of the reduced TED must be unique.*

It is possible to further reduce the graph by performing *normalization*, similar to the one described for *BMDs [2]. The normalization procedure starts by moving the numeric values from the non-zero terminal nodes to the terminal edges, where they are assigned as edge *weights*. This is shown in Fig. 11(d) and Fig. 13(b). By doing this, the terminal node holds constant 1. This operation applies to all terminal edges with terminal nodes holding values different than 1 or 0. As a result, only terminal nodes 1 and 0 are needed in the graph. The weights at the terminal edges may by further propagated to the upper edges of the graph, depending on their relative values. The TED normalization process that accomplishes this is defined as follows.

Definition 4.4 *A reduced, ordered TED representation is* **normalized** *when:*

- *The weights assigned to the edges spanning out of a given node are relatively prime.*
- *Numeric value 0 appears only in the terminal nodes.*
- *The graph contains no more than two terminal nodes, one each for 0 and 1.*

By ensuring that the weights assigned to the edges spanning out of a node are relatively prime, the extraction of common subgraphs is enabled. Enforcing the rule that none of the edges be allowed zero weight is required for the canonization of the diagram. When all the edge weights have been propagated up to the edges, only the value 0 and 1 can reside in the terminal nodes.

The normalization of the TED representation is illustrated by an example in Fig. 13. First, as shown in Fig. 13(b), the constants (6, 5) are moved from terminal nodes to terminal edges. These

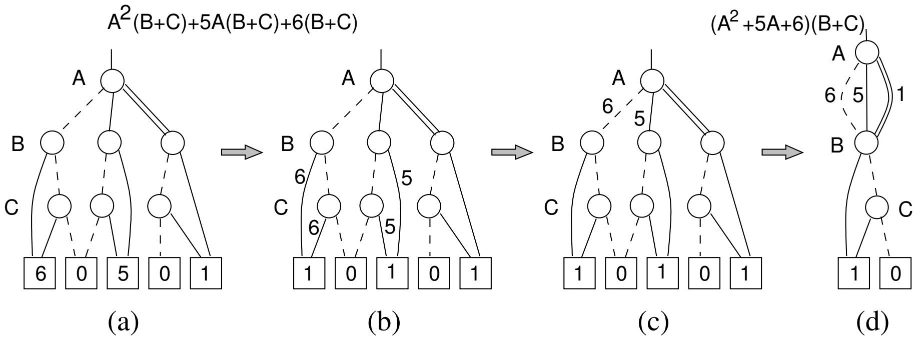


Figure 13: Normalization of the TED for $F = (A^2 + 5A + 6)(B + C)$

weights are then propagated up along the linear edges to the edges rooted at nodes associated with variable B, see Fig. 13(c). At this point the isomorphic subgraphs $(B + C)$ are identified at the nodes of B and the graph is subsequently reduced by merging the isomorphic subgraphs, as shown in Fig. 13(d).

It can be shown that normalization operation can reduce the size of a TED exponentially. Conversely, transforming a normalized TED to a non-normalized TED can, in the worst-case, result in an exponential increase in the graph size. This result follows directly from the concepts of normalization of BMDs to *BMDs [2].

4.5 Canonicity of Taylor Expansion Diagrams

It now remains to be shown that an ordered, reduced and normalized Taylor Expansion Diagram is canonical; *i.e.* for a fixed ordering of variables, any algebraic expression is represented by a unique reduced, ordered and normalized TED. First, we recall the following Taylor's Theorem, proved in [68].

Theorem 1 *(Taylor's Theorem [68]) Let $f(x)$ be a polynomial function in the domain R, and let $x = x_0$ be any point in R. There exists one and only one unique Taylor's series with center x_0 that represents $f(x)$ according to the equation 17.*

The above theorem states the uniqueness of the Taylor's series representation of a function, evaluated at a particular point (in our case at $x = 0$). This is a direct consequence of the fact that the successive derivatives of a function evaluated at a point are unique. Using the Taylor's theorem and the properties of reduced and normalized TEDs, it can be shown that an ordered, reduced and normalized TED is canonical.

Theorem 2 *For any multivariate polynomial f with integer coefficients, there is a unique (up to isomorphism) ordered, reduced and normalized Taylor Expansion Diagram denoting f, and any other Taylor Expansion Diagram for f contains more vertices. In other words, an ordered, reduced and normalized TED is minimal and canonical.*

Proof: The proof of this theorem follows directly the arguments used to prove the canonicity and minimality of BDDs [1] and *BMDs [2].

Uniqueness. First, a reduced TED has no trivial redundancies; the redundant nodes are eliminated by the reduce operation. Similarly, a reduced TED does not contain any isomorphic subgraphs. Moreover, after the normalization step, all common subexpressions are shared by further application of the reduce operation. By virtue of the Taylor's Theorem all the nodes in an ordered, reduced and normalized TED are unique and distinguished.
Canonicity. We now show that the individual Taylor expansion terms, evaluated recursively, are uniquely represented by the internal nodes of the TED. First, for polynomial functions the Taylor series expansion at a given point is finite and, according to the Taylor's Theorem, the series is unique. Moreover, each term in the Taylor's series corresponds to the successive derivatives of the function evaluated at that point. By definition, the derivative of a differentiable function evaluated at a particular point is also unique. Since the nodes in the TED correspond to the recursively computed derivatives, every node in the diagram uniquely represents the function computed at that node. Since every node in an ordered, reduced and normalized TED is distinguished and it uniquely represents a function, the Taylor Expansion Diagram is canonical.

Minimality. We now show that a reduced, ordered and normalized TED is also minimal. This can be proved by contradiction. Let G be a graph corresponding to a reduced, normalized and hence canonical TED representation of a function f. Assume there exists another graph G', with the same variable order as in G, representing f that is smaller in size than G. This would imply that graph G could be reduced to G' by the application of reduce and normalize operations. However, this is not possible as G is a reduced and normalized representation and contains no redundancies. The sharing of identical terms across different decomposition levels in the graph G has been captured by the reduction operation. Thus G' cannot have a representation for f with fewer nodes than G. Hence G is a minimal and canonical representation for f. [Q.E.D.]

4.6 Complexity of Taylor Expansion Diagrams

Let us now analyze the worst-case size complexity of an ordered and reduced Taylor Expansion Diagram. For a polynomial function of degree k, decomposition with respect to a variable can produce k+1 *distinct* Taylor expansion terms in the worst-case.

Theorem 3 *Let f be a polynomial in n variables and maximum degree k. In the worst case, the ordered, reduced, normalized Taylor Expansion Diagram for f requires $O(k^{n-1})$ nodes and $O(k^n)$ edges.*

Proof: The top-level contains only one node, corresponding to the first variable. Since its maximum degree is k, the number of distinct children nodes at the second level is bounded by $k+1$. Similarly, each of the nodes at this level produces up to $k+1$ children nodes at the next level, giving a rise to $(k+1)^2$ nodes, and so on. In the worst case the number of children increases in geometric progression, with the level i containing up to $(k+1)^{i-1}$ nodes. For an n-variable function, there will be n-1 such levels, with the n-th level containing just two terminal nodes, 1 and 0. Hence the total number of internal nodes in the graph is $N = \sum_{i=0}^{n-1}(k+1)^i = \frac{(k+1)^n-1}{k}$. The number of edges E can be similarly computed as $E = \sum_{i=1}^{n}(k+1)^i = \frac{(k+1)^{n+1}-1}{k} - 1$, since there may be up to $(k+1)^n$ terminal edges leading to the 0 and 1 nodes. Thus, in the worst-case, the total number of internal nodes required to represent an n-variable polynomial with degree k is $O(k^{n-1})$ and the number of edges is $O(k^n)$. [Q.E.D.]

One should keep in mind, however, that the TED variables represent symbolic, word-level signals, and the number of such signals in the design is significantly smaller than the number of bits in the bit-level representation. Subsequently, even an exponential size of the polynomial with a relatively small number of such variables may be acceptable. Moreover, for many practical designs the complexity is not exponential.

Finally, let us consider the TED representation for functions with variables encoded as n-bit vectors, $X = \sum_{i=0}^{n-1} 2^i x_i$. For linear expressions, the space complexity of TED is linear in the number of bits n, the same as *BMD. For polynomials of degree $k \geq 2$, such as X^2, etc., the size of *BMD representation grows polynomially with the number of bits, as $O(n^k)$. For K*BMD the representation also becomes nonlinear, with complexity $O(n^{k-1})$, for polynomials of degree $k \geq 3$. However, for ordered, reduced and normalized TEDs, the graph remains linear in the number of bits, namely $O(n \cdot k)$, for any degree k, as stated in the following theorem.

Theorem 4 *Consider variable X encoded as an n-bit vector, $X = \sum_{i=0}^{n-1} 2^i x_i$. The number of internal TED nodes required to represent X^k in terms of bits x_i, is $k(n-1)+1$.*

Proof: We shall first illustrate it for the quadratic case $k = 2$. Let W_n be an n-bit representation of X: $X = W_n = \sum_{i=0}^{n-1} 2^i x_i = 2^{(n-1)} x_{n-1} + W_{n-1}$ where $W_{n-1} = \sum_{i=0}^{n-2} 2^i x_i$ is the part of X containing the lower (n-1) bits. With that, $W_n^2 = (2^{n-1} x_{n-1} + W_{n-1})^2 = 2^{2(n-1)} x_{n-1}^2 + 2^n x_{n-1} W_{n-1} + W_{n-1}^2$. Furthermore, let $W_{n-1} = (2^{n-2} x_{n-2} + W_{n-2})$, and $W_{n-1}^2 = (2^{2(n-2)} x_{n-2}^2 + 2^{n-1} x_{n-2} W_{n-2} + W_{n-2}^2)$.

Notice that the constant term (0-edge) of W_{n-1} w.r.to variable x_{n-2} contains the term W_{n-2}, while the linear term (1-edge) of W_{n-1}^2 contains $2^{n-1} W_{n-2}$. This means that the term W_{n-2} can be *shared* at this decomposition level by two different parents.As a result, there are exactly two non-constant terms, W_{n-2} and W_{n-2}^2 at this level.

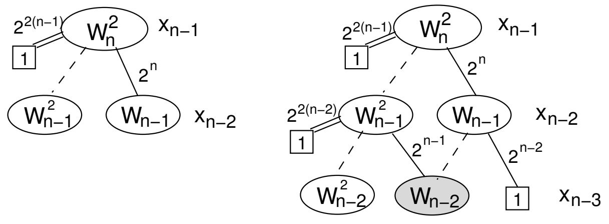


Figure 14: Construction of TED for X^2 with n bits

In general, at any level l, associated with variable x_{n-l}, the expansion of terms W_{n-l}^2 and W_{n-l} will create *exactly two* different non-constant terms, one representing W_{n-l-1}^2 and the other W_{n-l-1}; plus a constant term 2^{n-l}. The term W_{n-l-l} will be shared, with different multiplicative constants, by W_{n-l}^2 and W_{n-l}.

This reasoning can be readily generalized to arbitrary integer degree k; at each level there will always be exactly k different non-constant terms. Since on the top variable (x_{n-1}) level there is only one node (the root), and there are exactly k non-constant nodes at each of the remaining $(n-1)$ levels, the total number of nodes is equal to $k(n-1)+1$. □

The derivation of TED representation for X^2 generalized to n bits is shown in Figure 15,

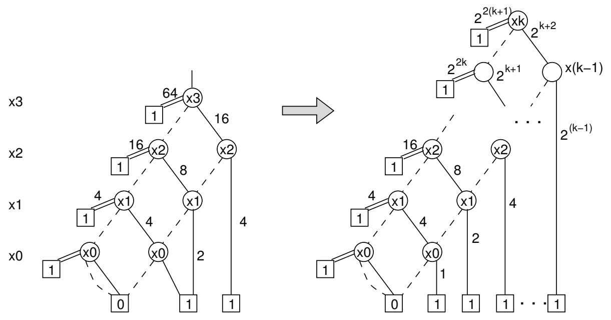


Figure 15: Derivation of TED representation for X^2 with n bits

Table 1 compares the worst-case size complexity of the canonical "decision" diagrams described in this chapter in terms of the number of nodes as a function of the size of their operands (bit-width n). It shows significantly lower worst-case complexity of TED compared to other representations.

Table 1: Size complexity of different canonical diagrams

Diagram type	X	$X+Y$	$X \cdot Y$	X^k	c^X
MTBDD	exp	exp	exp	exp	exp
EVBDD	lin	lin	exp	exp	exp
*BMD	lin	lin	lin	n^k	lin
K*BMD	lin	lin	lin	n^{k-1}	lin
TED	const	const	const	$(n-1)k$	–

4.7 Composition of Taylor Expansion Diagrams

Taylor Expansion Diagrams can be composed to compute complex expressions from simpler ones. This section describes general composition rules to compute a new TED as an algebraic sum (+) or product ($\cdot$) of two TEDs. The general composition process for TEDs is similar to that of the APPLY operator for BDD's [1], in the sense that the operations are recursively applied on respective graphs. However, the composition rules for TEDs are specific to the rules of the algebra $(R, \cdot, +)$.

Starting from the roots of the two TEDs, the TED of the result is constructed by recursively constructing all the non-zero terms from the two functions, and combining them, according to a given operation, to form the diagram for the new function. To ensure that the newly generated nodes are unique and minimal, the REDUCE operator is applied to remove any redundancies in the graph.

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. The top node q of the resulting TED is associated with the variable with the higher order, i.e., $var(q) = x$, if $x \geq y$, and $var(q) = y$ otherwise. Let f, g be two functions rooted at nodes u, v, respectively, and h be a function rooted at the new node q.

For the purpose of illustration, we describe the operations on linear expressions, but the analysis is equally applicable to polynomials of arbitrary degree. In constructing these basic operators, we must consider several cases:

1. Both nodes u, v are terminal nodes. In this case a new *terminal* node q is created as $val(q) = val(u) + val(v)$ for the ADD operation, and as $val(q) = val(u) \cdot val(v)$ for the MULT operation.
2. At least one of the nodes is non-terminal. In this case the TED construction proceeds according to the variable order. Two cases need to be considered here: (a) when the top nodes u, v have the same index, and (b) when they have different indices. The detailed analysis of both cases is given in [70]. Here we show the multiplication of two diagrams rooted at variables u and v with the same index.

   $$\begin{aligned} h(x) &= f(x) \cdot g(x) = (f(0) + xf'(0)) \cdot (g(0) + xg'(0)) \\ &= [f(0)g(0)] + x[f(0)g'(0) + f'(0)g(0)] + x^2[f'(0)g'(0)]. \end{aligned} \tag{19}$$

   In this case, the 0-child of q is obtained by pairing the 0-children of u, v. Its 1-child is created as a sum of two cross products of 0- and 1-children, thus requiring an additional ADD operation. Also, an additional 2-child (representing the quadratic term) is created by pairing the 1-children of u, v.

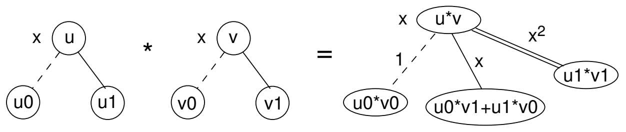


Figure 16: Multiplicative composition for nodes with same variables.

Figure 17 illustrates the application of the ADD and MULT procedures to two TEDs. As shown in the figure, the root nodes of the two TEDs have the same variable index. The MULT operation requires the following steps: (i) performing the multiplication of their respective constant (0-) and linear (1-) children nodes; and (ii) generating the sum of the cross-products of their 0- and 1-children. On the other hand, the two TEDs corresponding to the resulting cross-product, as highlighted in the figure, have different variable indices for their root nodes. In this case, the node with the lower index corresponding to variable C is added to the 0-child of the node corresponding to variable B.

It should be noted that the ADD and MULT procedures described above will initially produce non-normalized TEDs, with numeric values residing only in the terminal nodes, requiring further normalization. When these operations are performed on normalized TEDs, with weights assigned to the edges, then the following modification is required: when the variable indices of the root nodes of f and g are different, the edge weights have to be propagated down to the children nodes recursively. Downward propagation of edge weights results in the dynamic update of the edge weights of the children nodes. In each recursion step, this propagation of edge weights down to the children proceeds until the weights reach the terminal nodes. The numeric values are updated only in the terminal nodes. Every time a new node is created, the REDUCE and NORMALIZE operations are required to be performed in order to remove any redundancies from the graph and generate a minimal and canonical representation.

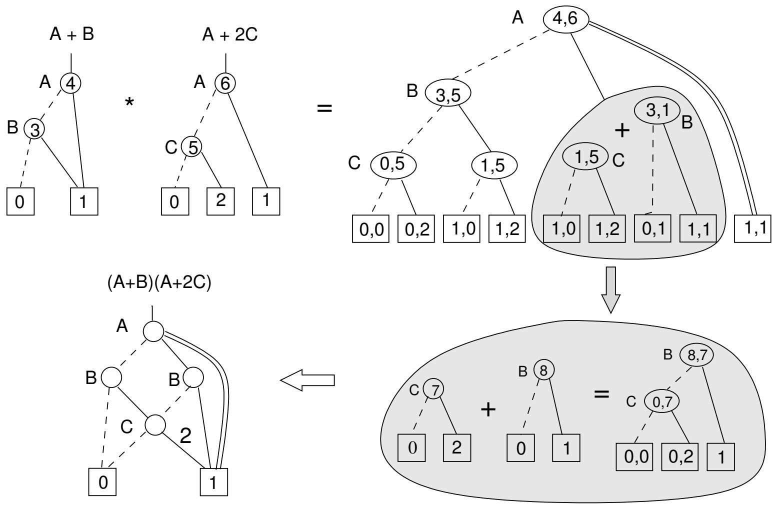


Figure 17: Example of MULT composition: (A+B)(A+2C).

4.8 Design Modeling and Verfication with TEDs

Using the operations described in the previous section, Taylor Expansion Diagrams can be constructed to represent various computations over integers in a compact, canonical form. The compositional operators ADD and MULT can be used to compute any combination of arithmetic functions by operating directly on their TEDs. However, the representation of Boolean logic, often present in the RTL designs, requires special attention since the output of a logic block must evaluate to Boolean rather than to an integer value.

Boolean Logic. Similarly to *BMDs, one can also define TED operators for Boolean logic, OR, AND, and XOR, where both the range and domain of function are Boolean. This can be done in much the same way as it is done for *BMDs. In fact, the TED and *BMD for a Boolean logic are identical, because they require only the first moment decomposition (refer to Figure 5).

Similarly one can derive other operators which rely on Boolean variables as one of their inputs, with other inputs being word-level. One such example is the multiplexer, $\text{MUX}(c,X,Y) = c \cdot X + (1-c) \cdot Y$, where c is a binary control signal, and X and Y are word-level inputs.

In general, TED, which represents an integer-valued function, will also correctly model designs with arithmetic and Boolean functions. Note that the ADD (+) function will always create correct integer result over Boolean and integer domains, because Boolean variables are treated as binary (0,1), a special case of integer. However the MULT $(\cdot)$ function may create powers of Boolean variables, x^k, which should be reduced to x. A minor modification of TED is done to account for this effect so that the Boolean nature of variable x can be maintained in the representation. Such modified Taylor Expansion Diagrams are also canonical.

TED Construction for RTL Designs. TED construction for an RTL design starts with building trivial TEDs for its primary inputs. Partial expansion of the word-level input signals is often necessary when one or more bits from any of the input signals fan out to other parts of the design. This is the case in the designs shown in Fig. 18 (a) and (b), where bits $a_k = A[k]$ and $b_k = B[k]$ are derived from word-level variables A and B. In this case, the word-level variables must be decomposed into several word-level variables with shorter bit-widths. In our case, $A = 2^{(k+1)}A_{hi} + 2^k a_k + A_{lo}$ and $B = 2^{(k+1)}B_{hi} + 2^k b_k + B_{lo}$, where $A_{hi} = A[n-1{:}k{+}1]$, $a_k = A[k]$, and $A_{lo} = A[k\text{-}1{:}0]$; and similarly for variable B. Variables $A_{hi}, a_k, A_{lo}, B_{hi}, b_k, B_{lo}$ form the *abstracted* primary inputs of the system. The basic TEDs are readily generated for these abstracted inputs from their respective bases (A_{hi}, a_k, A_{lo}), and (B_{hi}, b_k, B_{lo}).

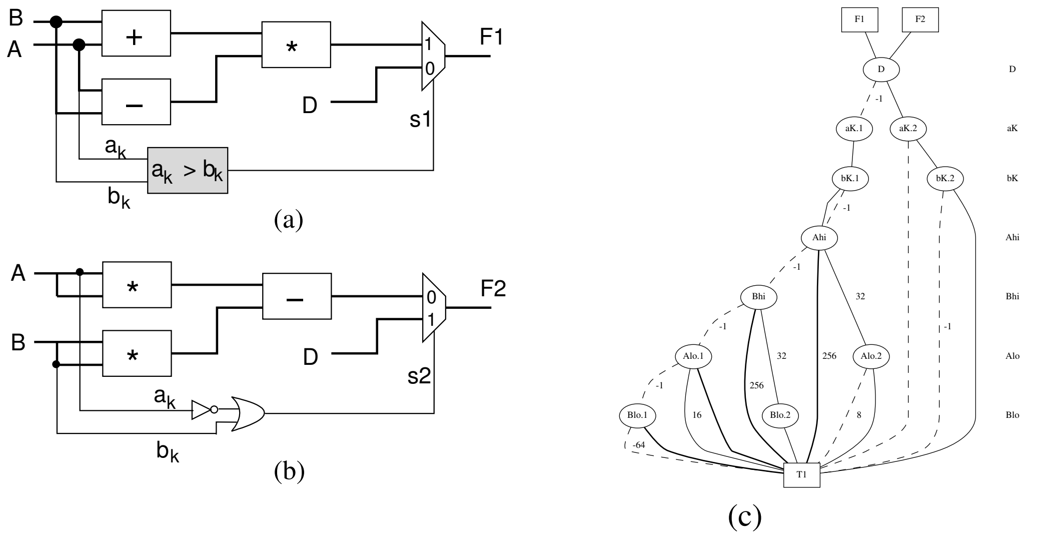


Figure 18: RTL verification using canonical TED representation: (a), (b) Functionally equivalent RTL modules; (c) The isomorphic TED for the two designs.

Once all the abstracted primary inputs are represented by their TEDs, Taylor Expansion Diagrams can be constructed for all the components of the design. TEDs for the primary outputs are then generated by systematically composing the constituent TEDs in the topological order, from the primary inputs to the primary outputs. For example, to compute $A + B$ in Fig. 18 (a) and (b), the ADD operator is applied to functions A and B (each represented in terms of their abstracted components). The subtract operation, $A - B$, is computed by first multiplying B with a constant -1 and adding the result to the TED of A. The multipliers are constructed from their respective inputs using the MULT operator, and so on. To generate a TED for the output of the multiplexors, the Boolean functions s_1 and s_2 first need to be constructed as TEDs. Function s_1 is computed by transforming the single-bit comparator $a_k > b_k$ into a Boolean function and expressed as an algebraic equation, $s_1 = a_k \wedge \overline{b_k} = a_k \cdot (1 - b_k)$, as described in Section 4.8. Similarly, $s_2 = \overline{a_k} \vee b_k$ is computed as $s_2 = 1 - a_k \cdot (1 - b_k)$ and represented as a TED. Finally, the TEDs for the primary outputs are generated using the MUX operator with the respective inputs. As a result of such a series of composition operations, the outputs of the TED represent multi-variate polynomials in terms of the primary inputs of the design.

TED-based Verification. After having constructed the respective ordered, reduced, and normalized Taylor Expansion Diagram for each design, the test for functional equivalence is performed by checking for isomorphism of the resulting graphs. In fact, the TED-based verification is similar to that using BDDs and BMDs: the generation of the TEDs for the two designs under verification takes place in the same TED manager; when the two functions are equivalent, both functions point to the same root of the common TED. This is shown in Fig. 18(c).

It should be noted that the arithmetic operations in these designs assume that no overflow is produced by any of the intermediate signal. That is, functions $F1$ and $F2$ are functionally equivalent under *infinite precision computation* model. This limitaion is a natural consequence of the design representation on the abstract level, where notion of the individual bits is not available.

4.9 Implementation and Experimental Results

A prototype version of TED software for behavioral HDL designs has been implemented using as a front end a popular high-level synthesis system GAUT [71]. This system was selected due to its commercial quality, robustness, and its open architecture. The input to the system is behavioral VHDL or C description of the design. The design is parsed and the extracted data flow is automatically transformed into canonical TED representation.

The core computational platform of the TED package consists of a *manager* that performs the construction and manipulation of the graph. It provides routines to uniquely store and manipulate the nodes, edges and terminal values, in order to keep the diagrams canonical. To support canonicity, the nodes are stored in a hash table, implemented as *unique table*, similar to that of the CUDD package [14], [72]. The table contains a *key* for each vertex of the TED, computed from the node index and the attributes of its children and the edge weights. As a result, equivalence test between two TEDs reduces to a simple scalar test between the identifiers of the corresponding vertices.

Variable Ordering. Since TEDs are a canonical representation subject to the imposition of a total ordering on the variables, it is desirable to search for a variable order that would minimize the size of TEDs. Dynamic variable ordering for TEDs is based on local swapping of adjacent variables in the diagram, similar to those employed in BDD ordering [73, 74]. It has been shown that, similarly to BDDs, local swapping of adjacent variables does not affect the structure of the diagram outside of the swapping area.

In addition, TEDs can be subjected to static ordering. Typically, the variables are ordered topologically, from primary inputs to primary outputs, in the order in which the signals appear in the design specification. Coefficients are usually represented as weights associated with TED edges. In some cases, however, it may be beneficial to treat some of the coefficients as special variables, rather than weights associated with edges, and place them in the TED graph above all the signal variables. This is particularly important when TEDs are used for the purpose of expression simplification and TED decomposition, as it facilitates symbolic factorization and common subexpression elimination [75].

Experimental Setup. Several experiments were performed using the prototype TEDify software on a number of dataflow designs described in behavioral VHDL. The designs range from simple algebraic (polynomial) computations to those encountered in signal and image processing algorithms. Simple RTL designs with Boolean-algebraic interface were also tested.

The experiments with TED were conducted as follows. The design described in behavioral VHDL or C was parsed by a high-level synthesis system GAUT [71]. The extracted data flow was then automatically translated into a canonical TED representation using the experimental software TEDify. Comparisons against *BMDs were conducted to demonstrate the power of abstraction of TED representation. For this purpose, each design was synthesized into a structural netlist from which *BMDs were constructed. In most cases BDDs could not be constructed due to their prohibitive size, and they are not reported. Experiments confirm that word-size abstraction by TEDs results in much smaller graph size and computation times as compared to *BMDs.

Verification of High-level Transformations. During the process of architectural synthesis, the initial HDL description often proceeds through a series of high-level transformations. For example, computation $AC + BC$ can be transformed into an equivalent one, $(A+B)C$, which better utilizes the hardware resources. TEDs are ideally suited to verify the correctness of such transformations by proving equivalence of the two expressions, regardless of the word size of the input/output signals. We performed numerous experiments to verify the equivalence of such algebraic expressions. Results indicate that both time and memory usage required by TEDs is orders of magnitude smaller as compared to *BMDs. For example, the expression $(A+B)(C+D)$, where A, B, C, D are n-bit vectors, has a TED representation containing just 4 internal nodes, regardless of the word size. The size of *BMD for this expression varies from 418 nodes for the 8-bit vectors, to 2,808 nodes for 32-bit variables. BDD graphs could not be constructed for more than 15 bits.

RTL Verification. As mentioned earlier, TEDs offer the flexibility of representing designs containing both arithmetic operators and Boolean logic. We used the generic designs of Figure 18 and performed a set of experiments to observe the efficiency of TED representation under varying size of Boolean logic. The size of the algebraic signals A, B was kept constant at 32 bits, while the word size of the comparator (or the equivalent Boolean logic) was varied from 1 to 20. As the size of Boolean logic present in the design increases, the number of bits extracted from A, B also increases (the figure shows it for single bits). Table 2 gives the results obtained with TED and compares them to those of *BMDs. Note that, as the size of Boolean logic increases, TED size converges to that of *BMD. This is to be expected as *BMDs can be considered as a special (Boolean) case of TEDs.

Table 2: Size of TED vs. Boolean logic

bits	*BMD		TED	
(k)	Size	CPU time	Size	CPU
4	4620	107 s	194	44 s
8	15K	87 s	998	74 s
12	19K	93 s	999	92 s
16	23.9K	249 s	4454	104 s
18	timeout	>12 hrs	12.8K	29 min
20	timeout	>12 hrs	timeout	>12 hrs

Array Processing. An experiment was also performed to analyze the capability of TEDs to represent computations performed by an array of processors. The design that was analyzed is an $n \times n$ array of configurable Processing Elements (PE), which is a part of a low power motion estimation architecture [76]. Each processing element can perform two types of computations on a pair of 8-bit vectors, A_i, B_i, namely $(A_i - B_j)$ or $(A_i^2 - B_j^2)$, and the final result of all PEs is then added together. The size of the array was varied from 4×4 to 16×16, and the TED for the final result was constructed for each configuration.

When the PEs are configured to perform subtraction $(A_i - B_j)$, both TEDs and *BMDs can be constructed for the design. However, when the PEs are configured to compute $A_i^2 - B_j^2$, the size of *BMDs grows quadratically. As a result, we were unable to construct *BMDs for the 16×16 array of 8-bit processors. In contrast, the TEDs were constructed easily for all the cases. The results are shown in Table 3. Note that we were unable to construct the BDDs for any size n of the array for the quadratic computation.

Table 3: PE computation: $(A_i^2 - B_j^2)$.

Array size	*BMD		TED	
$(n \times n)$	Size	CPU time	Size	CPU time
4×4	123	3 s	10	1.2 s
6×6	986	3.4 s	14	1.5 s
8×8	6842	112 s	18	1.6 s
16×16	out of mem	-	34	8.8 s

DSP Computations. One of the most suitable applications for TED representation are algorithmic descriptions of dataflow computations, such as digital signal and image processing algorithms. For this reason, we have experimented with the designs that implement various DSP algorithms.

Table 4 presents some data related to the complexity of the TEDs constructed for these designs. The first column in the Table describes the computation implemented by the design. These include: FIR and IIR filters, fast Fourier transform (FFT), elliptical wave filter ($Elliptic$), least mean square computation ($LMS128$), discrete cosine transform (DCT), matrix product computation ($ProdMat$), Kalman filter ($Kalman$), etc. Most of these designs perform algebraic computations by operating on vectors of data, which can be of arbitrary size. The next column gives the number of inputs for each design. While each input is a 16-bit vector, TED represents them as word-level symbolic variable. Similarly, the next column depicts the number of 16-bit outputs. The remaining columns of the table show: BMD size (number of nodes), CPU time required to construct the BMD for the 16-bit output words, TED size (number of nodes) required to represent the entire design; CPU times required to generate TED diagrams does not account for the parsing time of the GAUT front end.

Table 4: Signal Processing Applications

Design	Input size	Output size	*BMD size (nodes)	BMD CPU time (s)	TED size (nodes)	TED CPU time (s)
Dup-real	3x16	1x16	92	10	5	1
IIR	5x16	1x16	162	13	7	1
FIR16	16x16	1x16	450	25	18	1
FFT	10x16	8x16	995	31	29	1
Elliptic	8x16	8x16	922	19	47	1
LMS128	50x16	1x16	8194	128	52	1
DCT	32x16	16x16	2562	77	82	1
ProdMat	32x16	16x16	2786	51	89	1
Kalman	77x16	4x16	4866	109	98	1

Fig. 19 depicts a multiple-output TED for the elliptical wave filter (design *elliptic*), where each root node corresponds to an output of the design.

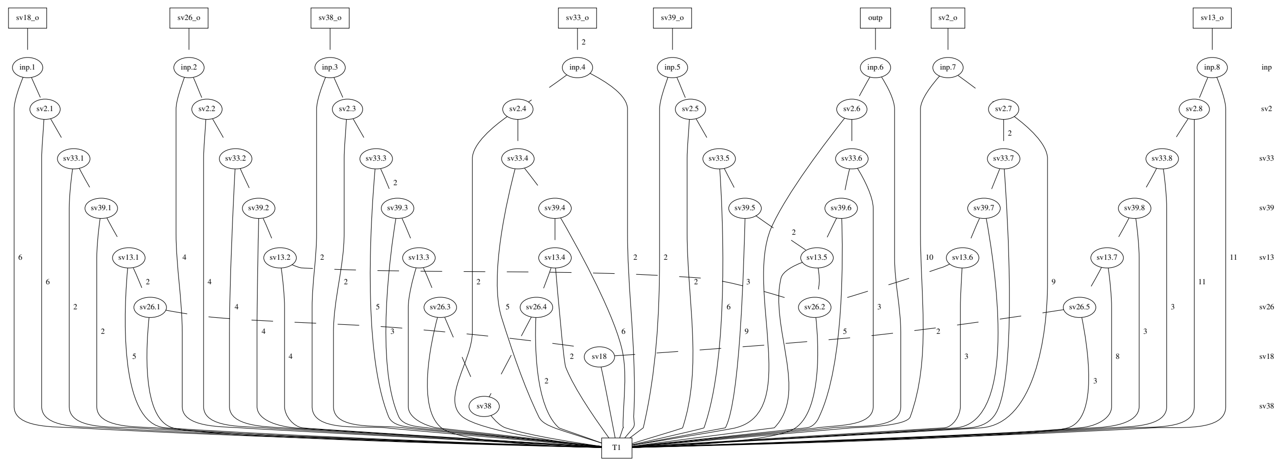


Figure 19: Elliptic Wave Filter: TED structure obtained automatically from VHDL description.

Algorithmic Verification. This final set of experiments demonstrates the natural capability of Taylor Expansion Diagrams to verify equivalence of designs described at the *algorithmic* level. Consider two dataflow designs computing convolution of two real vectors, $A(i), B(i), i = 0, \ldots N-1$, shown in Fig. 20. The design in Fig. 20(a) computes FFT of each vector, computes product of the FFT results, and performs the inverse FFT operation, producing output $IFFT$. The operation shown in Fig. 20(b) computes convolution directly from the two inputs, $C(i) = \sum_{k=0}^{N-1} A(k) \cdot B(i-k)$. TED was used to represent these two computations for $N = 4$ and to prove that they are indeed equivalent. Fig. 21 depicts the TED for vector C of the convolution operation, isomorphic with the vector $IFFT$. All graphs are automatically generated by our TED-based verification software.

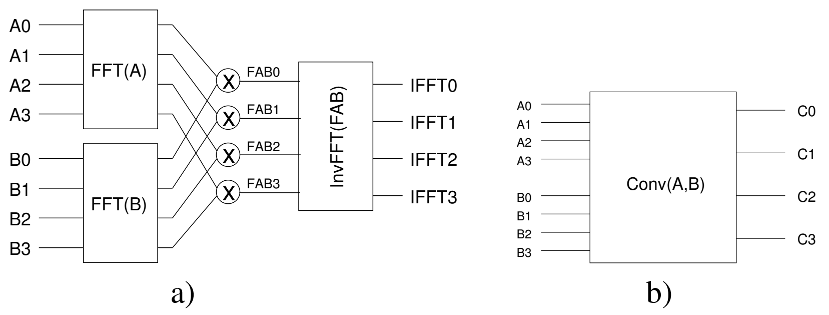


Figure 20: Equivalent Computations: (a) FFT-Product-Inv(FFT); (b) Convolution.

As illustrated by the above example, TEDs can be suitably augmented to represent computations in the complex domain. In fact, it can be shown that TEDs can represent polynomial functions over an *arbitrary field*. The only modification required is that the weights on the graph edges be elements of the field, and that the composition (MULT and ADD) be performed with the respective operators of the field. Subsequently, TEDs can also be used to represent computations in *Galois field* [77].

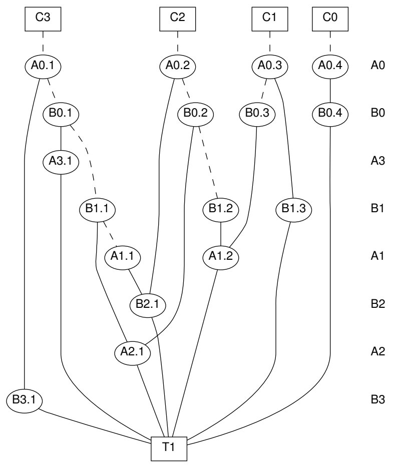


Figure 21: TED for convolution vector C, isomorphic with $IFFT$

4.10 Limitations of TED Representation

TEDs have several natural limitations. As mentioned eralier, TEDs can only be used to represent infinite precision arithmetic, and cannot represent modular arithmetic. Furthermore, they can only represent functions that have *finite* Taylor expansion, and in particular multi-variate polynomials with finite integer degrees. For polynomials of finite integer degree $k \geq 1$, successive differentiation of the function ultimately leads to zero, resulting in a finite number of terms. However, those functions that have infinite Taylor series (such as a^x, where a is a constant) cannot be represented with a finite TED graph. In order to represent exponentials using TEDs, one must expand the integer variable into bits, $X = \{x_{n-1}, x_{n-2}, \ldots, x_0\}$, and use the TED formulas to represent the function in terms of the bits. Such a TED would be structurally similar to the *BMD representation of the function.

While TED representation naturally applies to functions that can be modeled as *finite polynomials*, the efficiency of TED relies on its ability to encode the design in terms of its *word-level* symbolic inputs, rather than bit-level signals. This is the case with the simple RTL designs shown in Figure 18, where all input variables and internal signals have simple, low-degree polynomial representation. The abstracted word-level inputs of these designs are created by partial bit selection (a_k, b_k) at the primary inputs, and a polynomial function can be constructed for its outputs. However, if any of the internal or output signals is partitioned into sub-vectors, such sub-vectors cannot be represented as polynomials in terms of the symbolic, word-level input variables, but depend on the individual bits of the inputs. The presence of such signal splits creates a fundamental problem for the polynomial representations, and TEDs cannot be used efficiently in those cases. For similar reasons TED cannot represent modular arithmetic. An attempt to fix this problem was proposed in [77], by modeling the discrete functions as finite, word-level polynomials in Galois Field (GF). The resulting polynomials, however, tend to be of much higher degree than the original function, with the degree depending on the signal bit-width, making the representation less efficient for practical applications. This is the case where TEDs can exhibit space explosion similar to that encountered in BDDs and BMDs.

Another limitation of TEDs is that they cannot represent relational operators (such as comparators,

$A \geq B, A == B$, etc.) in symbolic form. This is because Taylor series expansion is defined for functions and not for relations. Relations are characterized by discontinuities over their domain and are not differentiable. In order to use TEDs to represent relational operators, often encountered in RTL descriptions, the expansion of word-level variables and bit vectors into their bit-level components is required.

Despite these limitation, TEDs can be successfully used for verifying equivalence of high-level, behavioral and algorithmic descriptions. Such algorithmic descriptions typically do not exhibit signal splits, hence resulting in polynomial functions over word-level variables.

4.11 Conclusions and Open Problems

This section described a compact, canonical, graph-based representation, called Taylor Expansion Diagram (TED). It has been shown that, for a fixed ordering of variables, TED is a canonical representation that can be used to verify equivalence of arithmetic computations in dataflow designs. It has been shown how TEDs can be constructed for behavioral and some RTL design descriptions. The power of abstraction of TEDs makes them particularly applicable to dataflow designs specified at the behavioral and algorithmic level.

For larger systems, especially those involving complex bit-select operations, and containing large portions of Boolean logic, relational operators and memories, TEDs can be used to represent those portions of the design that can be modeled as polynomials. Equivalence checking of such complex design typically involves finding structurally similar points of the designs under verification. TED data structure can be used here to raise the level of abstraction of large portions of designs, aiding in the identification of such similar points and in the overall verification process. In this sense TEDs complements existing representations, such as BDDs and *BMDs, in places where the level of abstraction can be raised.

The experiments demonstrate the applicability of TED representation to verification of dataflow designs specified at behavioral and algorithmic levels. This includes portions of algorithm-dominant designs, such as signal processing for multimedia applications and embedded systems. Computations performed by those designs can often be expressed as polynomials and be readily represented with TEDs. The test for functional equivalence is then performed by checking isomorphism of the resulting graphs. Of particular promise is the use of TEDs in verification of algorithmic descriptions, where the use of symbolic, word-level operands, without the need to specify their bit-width, is justified. A number of open problems remain to be researched to make TEDs a reliable data structure for high-level design representation and verification.

In addition to these verification-related applications, TEDs proves useful in algorithmic and behavioral synthesis and optimization for DSP and data-flow applications [75]. TED data structure, representing functional view of the computation can serve as an efficient vehicle to obtain a structural representation, namely the data flow graph (DFG). This can be obtained by means of graph decomposition which transforms the functional TED representation into a structural DFG representation. By properly guiding the decomposition process the resulting DFG can provide a better starting point for the ensuing architectural (high-level) synthesis, than the one extracted directly from the original HDL specification.

5 Represention of Multiple Output Functions Over Finite Fields

This section presents a method for representing multiple-output, binary and word-level functions in GF(N) ($N = p^m$, p a prime number and m a nonzero positive integer) based on decision diagrams (DD). The presented DD is canonical and can be made minimal with respect to a given variable order. The DD has been tested on benchmarks including integer multiplier circuits and the results show that it can produce better node compression (more than an order of magnitude in some cases) compared to shared BDDs. The benchmark results also reflect the effect of varying the input and output field sizes on the number of nodes. Methods of graph-based representation of characteristic and encoded characteristic functions in GF(N) are also presented. Performance of the proposed representations has been studied in terms of average path lengths and the actual evaluation times with 50,000 randomly generated patterns on many benchmark circuits. All these results reflect that the proposed technique can out perform existing techniques.

5.1 Previous Work

Finite fields have numerous applications in public-key cryptography [78] to encounter channel errors and for protection of information, error control codes [79] and digital signal processing [80]. Finite fields gained significance with practical lucrativeness of the elliptic curve crypto systems. The role of finite fields in error control systems is well established and contributes to many fault tolerant designs. In the EDA industry the role of multivalued functions, especially in the form of Multi-valued Decision Diagrams (MDD), is well described in [81,82]. Word-level diagrams can be useful in high level verification, logic synthesis [1, 83], and software synthesis [84]. Multivalued functions can also be represented in finite fields as shown in [85]. Finite fields can represents many arithmetic circuits very efficiently [86]. Also there are fine grain FPGA structures for which arithmetic circuits in finite fields seem to be highly efficient. The varied use of finite fields leads to designing high speed, low complexity systolic VLSI realizations [87]. Fast functional simulation in the design cycles is a key step in all these applications [88].

Most existing techniques for word-level representation, e.g. [2, 89], are not capable of efficiently representing arbitrary combinations of bits or nibbles, i.e. subvectors, within a word. The proposed framework for representing circuits can deal with these types of situations by treating each subvector as a word-level function in GF(2^m), where m denotes the number of bits within a subvector. The word-level functions are then represented as canonic word-level graphs. Hence the proposed technique offers a generalized framework for verifying arbitrary combinations of bits or words.

Another situation where existing word-level techniques seem to have difficulty is representing *non-linear* design blocks, such as comparators, at the register transfer level, RTL (e.g., in the integer domain). The proposed framework does not suffer from this critical shortcoming.

As an example of representing an arbitrary combination of output bits in a multiple-output function, lets consider a 4-input 8-output binary function. The MSB expressed by $f = \sum m(10,11,12,13,14,15)$ [1], and the LSB $g = \sum m(4,5,6,7,8,9,10,11,14,15)$ can be represented

[1]The notation $h = \sum m(p_1, p_2, \ldots, p_q)$ is used to represent the truth-table of a function where each p_r ($1 \leq r \leq q$) is the decimal equivalent of a row in the input-part of the table with an output of 1, i.e. each p_r is a *minterm* from the

on the same diagram as shown Fig. 22 [90]. The BDD based representation of this circuit will require larger number of nodes.

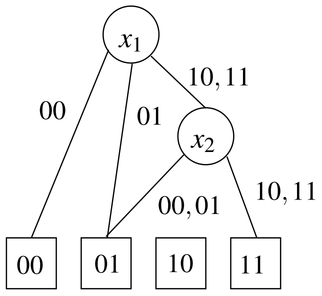


Figure 22: Representing two bits simultaneously.

Although research has been done on representing circuits in finite fields [86, 91], the theoretical basis was carried out, for example, in the spectral domain for a fixed value, e.g. 4 in [91]. Unlike these techniques, this section presents the generalized framework for the design, verification, and simulation of circuits in finite fields based on the MDD-like graph-based form. The proposed DD has advantages over other diagrams such as [2] in that, in addition to applications in multiple-valued algebra, it is not restricted to word boundaries, but instead it can be used to represent and verify any combination of output bits. Unlike [92], which is not a DD and hence lacks many features present in a DD, the proposed diagram does not have such shortcomings. Also, unlike [82], the proposed DD represents finite fields and extension fields, while [82] is based on the MIN-MAX post algebra.

Owing to its canonicity the proposed technique can be used for verifying circuits at the bit or word-level by checking for graph isomorphism, which can be done very quickly.

Fast evaluation times of multiple-output functions is significant in the areas of logic simulation, testing, satisfiability, and safety checking [93, 94]. The proposed DDs also offer much shorter average path lengths and hence evaluation times [95] compared to shared BDDs, with a varying trade-off between evaluation times and spatial complexity.

5.2 Background and Notation

Finite Fields Let GF(N) denote a set of N elements, where $N = p^m$ and p is a prime number and m a nonzero positive integer, with two special elements 0 and 1 representing the additive and multiplicative identities respectively, and two operators addition '$+$' and multiplication '$\cdot$'. GF(N) defines a finite field, also known as *Galois Field*, if it forms a *commutative ring* with identity over these two operators in which every element has a multiplicative inverse. In otherwords, GF(N) defines a finite field if the following properties hold.

- Associativity: $\forall a, b, c \in \mathrm{GF}(N)$ $(a+b)+c = a+(b+c)$, and $(a \cdot b) \cdot c = a \cdot (b \cdot c)$.
- Identity: $\forall a \in \mathrm{GF}(N)$ $a+0 = 0+a = a$, and $a \cdot 1 = 1 \cdot a = a$. Here '0' and '1' are the additive and multiplicative identities respectively.

ON-set in its decimal form.

- Inverse: $\forall a \in \mathrm{GF}(N)\ \exists -a, a^{-1} \in \mathrm{GF}(N)$ such that $a + (-a) = 0$ and $a \cdot a^{-1} = 1$. Here $-a$ and a^{-1} are the additive and multiplicative inverses respectively.
- Commutative: $\forall a, b \in \mathrm{GF}(N)\ a + b = b + a$, and $\forall c, d \in \mathrm{GF}(N) - \{0\}\ c \cdot d = d \cdot c$.
- Distributive: '$\cdot$' distributes over '$+$', i.e. $\forall a, b, c \in \mathrm{GF}(N)\ a \cdot (b + c) = (a \cdot b) + (a \cdot c)$.

Here p, which is a prime number, is called the characteristic of the field, and satisfies the following conditions: (a) $\underbrace{1 + 1 + \cdots + 1}_{p \text{ times}} = 0$. (b) $pa = 0$, $\forall a \in \mathrm{GF}(N)$. Also $\forall a \in \mathrm{GF}(N)$, $a^N = a$, and for $a \neq 0$, $a^{N-1} = 1$. The elements of $\mathrm{GF}(N)$ can be represented as polynomials over $\mathrm{GF}(p)$ of degree at most $n - 1$. There exists an element $\alpha \in \mathrm{GF}(N)$ for which the powers of $\alpha, \alpha^2, \ldots, \alpha^{N-1}$ are distinct and represent the nonzero elements of the field. Here α is called the *primitive element* of the field. Additional properties of $\mathrm{GF}(N)$ can be found in [79, 85].

Generation of Finite Fields A polynomial $p(x)$ over $\mathrm{GF}(p^m)$ is said to be primitive if it is *irreducible* (i.e. cannot be factored into lower degree polynomials), and if the smallest positive integer r for which $p(x)$ divides $x^r - 1$ is $r = p^m - 1$.

For example, the polynomial $p(x) = x^3 + x + 1$ is primitive over $\mathrm{GF}(2)$, because the smallest positive integer for which it is a divisor of $x^r - 1$ is $r = 7 = 2^3 - 1$, i.e. $x^7 - 1$.

Finite fields over $\mathrm{GF}(2^m)$ and $m \geq 2$ can be generated with primitive polynomials (PP) of the form $p(x) = x^m + \sum_{i=0}^{m-1} c_i x^i$, where $c_i \in \mathrm{GF}(2)$ [79].

For example given the PP $p(x) = x^3 + x + 1$, we can generate $\mathrm{GF}(8)$ as follows. Let α be a root of $p(x)$, i.e. $p(\alpha) = 0$. Hence $\alpha^3 + \alpha + 1 = 0$ or $\alpha^3 = \alpha + 1$. In general any element $\beta \in \mathrm{GF}(2^m)$ can be represented in this *polynomial form* as $\beta(x) = \sum_{i=0}^{m-1} \beta_i x^i$, where $\beta_i \in \{0, 1\}$. In this way all the elements of $\mathrm{GF}(8)$ can be generated as shown in Table 5. Note that since each coefficient $\beta_i \in \{0, 1\}$, each element in its polynomial form can also be represented as a bit vector, as shown in the third column. The bit vectors can be stored as integers in a computer program. For example, the element α^4 is 5 in decimal, which can be stored as an integer.

Operations Over Finite Fields For any $\alpha, \beta \in \mathrm{GF}(2^m)$, if α and β are in their polynomial forms as $\alpha(x) = \sum_{i=0}^{m-1} \alpha_i x^i$ and $\beta(x) = \sum_{i=0}^{m-1} \beta_i x^i$, where $\alpha_i, \beta_i \in \{0, 1\}$ and $0 \leq i < m$, then multiplication over $\mathrm{GF}(2^m)$ can be defined as $w(x) = \alpha(x) \cdot \beta(x) \bmod p(x)$, where $p(x)$ represents the PP used to generate the fields [79, 85, 96]. As an example, let $\alpha, \beta \in \mathrm{GF}(4)$, which is generated with the PP $p(x) = x^2 + x + 1$. Also let $\alpha(x) = x$ and $\beta(x) = x + 1$. Then

$$\begin{aligned} \alpha \cdot \beta &= \alpha(x) \cdot \beta(x) \bmod p(x) \\ &= x \cdot (x + 1) \bmod x^2 + x + 1 \\ &= x^2 + x \bmod x^2 + x + 1 \\ &= 1. \end{aligned}$$

If the elements are in their exponential form as in the first column of Table 5, then multiplications can also be carried out as follows. Let $a, b \in \mathrm{GF}(2^m)$ and that $a = \alpha^x$ and $b = \alpha^y$. Then $a \cdot$

Table 5: Generation of GF(2^3).

Exponential Representation		Polynomial Representation		Bit vector
0	$=$	0	$\leftrightarrow$	$[0,0,0]$
α^0	$=$	1	$\leftrightarrow$	$[0,0,1]$
α^1	$=$	α	$\leftrightarrow$	$[0,1,0]$
α^2	$=$	α^2	$\leftrightarrow$	$[1,0,0]$
α^3	$=$	$\alpha+1$	$\leftrightarrow$	$[0,1,1]$
α^4	$=$	$\alpha^2+\alpha$	$\leftrightarrow$	$[1,1,0]$
α^5	$=$	$\alpha^3+\alpha^2=\alpha^2+\alpha+1$	$\leftrightarrow$	$[1,1,1]$
α^6	$=$	α^2+1	$\leftrightarrow$	$[1,0,1]$

$b=\alpha^{(x+y) \bmod 2^m-1}$, where the addition is carried out over integers. For example, from Table 5 $\alpha^5\cdot\alpha^6=\alpha^{(5+6) \bmod 7}=\alpha^4$ over GF(8).

$\alpha+\beta$, i.e. addition over GF(2^m), is the bitwise EXOR of the bit vectors corresponding to $\alpha(x)$ and $\beta(x)$. For example, let $a=\alpha^5$ and $b=\alpha^6$ from Table 5. We also have $a=[1,1,1]$ and $b=[1,0,1]$. Hence $a+b=[0,1,0]$, which is α.

Fig. 23 shows multiplication and addition tables over GF(4).

$+$	0	1	α	β
0	0	1	α	β
1	1	0	β	α
α	α	β	0	1
β	β	α	1	0

(a) Addition

$\times$	0	1	α	β
0	0	0	0	0
1	0	1	α	β
α	0	α	β	1
β	0	β	1	α

(b) Multiplication

Figure 23: Addition and multiplication over GF(4).

Notation The following notation is used in this chapter.

Let $I_N=\{0,1,\dots,N-1\}$, and $\delta: I_N\rightarrow GF(N)$ be a one-to-one mapping, with $\delta(0)=0$.

Let $f|_{x_k=y}$, called the *cofactor* of f w.r.t. $x_k=y$, represent the fact that all occurrences of x_k within f is replaced with y, i.e. $f|_{x_k=y}=f(x_1,x_1,\dots,x_k=y,\dots,x_n)$. The notation $f|_{x_i=y_i,x_{i+1}=y_{i+1},\dots,x_{i+j}=y_{i+j}}$ (or just $f|_{y_i,y_{i+1},\dots,y_{i+j}}$ when the context is clear) will be used to represent the replacement of variables $x_i,x_{i+1},\dots,x_{i+j}$ with the values $y_i,y_{i+1},\dots,y_{i+j}$ respectively.

We shall use the notation $|A|$ to represent the total number of nodes in a graph A.

We have the following in GF(N).

Theorem 5 *A function $f(x_1, x_2, \ldots, x_k, \ldots, x_n)$ in GF(N) can be expanded as follows.*

$$f(x_1, x_2, \ldots, x_k, \ldots, x_n) = \sum_{e=0}^{N-1} g_e(x_k) f|_{x_k=\delta(e)}, \qquad (20)$$

where $g_e(x_k) = 1 - [x_k - \delta(e)]^{N-1}$.

Proof. The proof is done by perfect induction as follows. From the properties of GF(N) we have, for any $a \in$ GF(N) such that $a \neq 0$, $a^{N-1} = 1$. Now in Theorem 5 if $x_k = \delta(r)$, then $g_r(x_k) = 1$ for $r \in I_N$. Furthermore, $g_s(x_k) = 0$ $\forall s \in I_N$ and $s \neq r$. Therefore, only one term, namely $f(x_1, x_2, \ldots, \delta(r), \ldots, x_n)$, remains on the right-hand-side of Eq. (20), while all the remaining terms equate to zero. Hence the proof follows. [Q.E.D.]

In Theorem 5 $g_e(x_k)$ is called a *multiple-valued literal*[2] in GF(N). Theorem 5 is known as the *literal-based expansion* of functions in GF(N). Theorem 5 reduces to the *Shannon's expansion* over GF(2) as follows. If we put $N = 2$ in Eq. (20) then

$$\begin{aligned} f(x_1, x_2, \ldots, x_k, \ldots, x_n) &= \sum_{e=0}^{2-1} g_e(x_k) f|_{x_k=\delta(e)} \\ &= g_0(x_k) f|_{x=0} + g_1(x_k) f|_{x=1} \\ &= \bar{x}_x f|_{x=0} + x_k f|_{x=1}, \end{aligned}$$

where $\bar{y}$ represents that y appears in its complemented form.

The product of literals is called a *product-term* or just a *product*.

Two product-terms are said to be *disjoint* if their product in GF(N) equates to zero.

An expression in GF(N) constituting product-terms is said to be disjoint if all of its product terms are pairwise disjoint.

This chapter is organized as follows. Section 5.3 presents the theory behind the graph based representation and its reduction, with methods for additional node and path optimizations. Section 5.9 provides the theory behind representing functions in GF(N) in terms of graph-based characteristic and encoded characteristic functions. The proposed methods offer much shorter evaluation times than existing approaches and Section 5.10 provides a technique for calculation of the average path lengths for approximating the evaluation times for the proposed representations. The proposed technique has been tested on many benchmark circuits. Finally in Section 5.11 we present the experimental results.

5.3 Graph-Based Representation

Any function in GF(N) can be represented by means of a MDD-like [82] data structure. However, unlike traditional MDDs, which are used to represent functions in the MIN-MAX post algebra, algebra of finite fields needs considering. Although an MDD type of data structure has been used for

[2]The term 'literal' was chosen because in GF(2) this expression reduces to the traditional Boolean literal, i.e. it represents a variable or its complement (inverse).

representing functions in finite fields in [91], the underlying mathematical framework was considered for GF(4) only, and no generalization was proposed for higher order fields and their extensions and no experimental results were reported, even though it was reported that generalization can be made. Also, no technique seems to exist, which can further optimize an MDD-like representation of functions in GF(N) by zero terminal node suppression and normalization. It should be noted that the technique of [82] has used a type of edge negation based on modular arithmetic. However, modular arithmetic in the form considered in [82] does not naturally comply with extension fields. Since an MDD has been defined in terms of functions in the MIN-MAX post algebra, to distinguish between these two algebras the MDD-like representation of functions in finite fields will be called *(M)ultiple-(O)utput (D)ecision (D)iagrams* or MODDs. Hence, traditional MDDs result in a post algebraic MIN-MAX SOP form, while with the MODD a canonic polynomial expression in GF(N) can be obtained. As an example, the MODD of Fig. 27(a), which represents a 4-valued function with values in $\{0,1,\alpha,\beta\}$ (assuming $\alpha = 2$ and $\beta = 3$), yields the following expression in the MIN-MAX postalgebra:

$$\begin{aligned} f(x_1,x_2,x_3) \quad = \quad & \beta(x_1^{\beta}x_2^{\beta} \vee x_1^{\{1,\alpha\}}x_2^{\{1,\alpha,\beta\}}x_3^{\beta} \vee x_1^{\beta}x_2^{\{1,\alpha\}}x_3^{\beta} \vee x_1^{0}x_2^{\{1,\alpha\}}x_3^{\beta}) \vee \\ & \alpha(x_1^{\beta}x_2^{\{1,\alpha\}}x_3^{\alpha} \vee x_1^{\{1,\alpha\}}x_2^{\{1,\alpha,\beta\}}x_3^{\alpha} \vee x_1^{0}x_2^{\{1,\alpha\}}x_3^{\alpha}) \vee \\ & (x_1^{\beta}x_2^{\{1,\alpha\}}x_3^{1} \vee x_1^{\{1,\alpha\}}x_2^{\{1,\alpha,\beta\}}x_3^{1} \vee x_1^{0}x_2^{\{1,\alpha\}}x_3^{1}). \end{aligned}$$

Here the symbol '$\vee$' has been used to denote MAX. MIN is denoted by the product-like notation. The expression x_i^S, where $S \subseteq \{0,1,\alpha,\beta\}$, is a literal defined in the MIN-MAX postalgebra as $x_i^S = MAX\_VALUE$, where $MAX\_VALUE = \beta$ in this case, if $x_i \in S$; $x_i^S = 0$ otherwise. In contrast $x_i^S = 1$ if $x_i \in S$; $x_i^S = 0$ otherwise in GF(N) (Theorem 5). The following multivariate polynomial results from the MODD in GF(4) by application of Theorem 5 followed by expansion and rearranging the terms:

$$\begin{aligned} f(x_1,x_2,x_3) \quad = \quad & \beta x_1^3x_2^3 + \alpha x_1^2x_2^3 + x_1x_2^3 + \alpha x_1^3x_2^2 + x_1^2x_2^2 + \beta x_1x_2^2 + x_3x_2 + \beta x_1^2x_2 + \alpha x_1x_2 + \beta x_2^2x_3 \\ & + \alpha x_2x_3 + \beta x_1^2x_2^3x_3 + \alpha x_1x_2^3x_3 + \alpha x_1^2x_2^2x_3 + x_1x_2^2x_3 + x_1^2x_2x_3 + \beta x_1x_2x_3. \end{aligned}$$

Definition 1 (Decision Diagram) *A decision diagram in GF(N) is a rooted directed acyclic graph with a set of nodes V containing two types of nodes:* (a) *A set of N terminal nodes or leaves with out-degree zero, each one labeled with a $\delta(s)$ and $s \in I_N$. Each terminal node u is associated with an attribute value$(u) \in$ GF(N).* (b) *A set of non-terminal nodes, with out-degree of N. Each non-terminal node v is associated with an attribute var$(v) = x_i$ and $1 \leq i \leq n$, and another attribute child$_j(v) \in V$, $\forall j \in I_N$, which represents each of the children (direct successor) of v.*

The correspondence between a function in GF(N) and an MODD in GF(N) can be defined as follows.

Definition 2 (Recursive Expansion) *An MODD in GF(N) rooted at v denotes a function f^v in GF(N) defined recursively as follows:* (a) *If v is a terminal node, then $f^v = value(v)$, where $value(v) \in$ GF(N).* (b) *If v is a non-terminal node with var$(v) = x_i$, then f^v is the function*

$$f^v(x_1,x_2,\ldots,x_i,\ldots,x_n) = \sum_{e=0}^{N-1} g_e(x_i) f^{child_e(v)},$$

where $g_e(x_i) = 1 - [x_i - \delta(e)]^{N-1}$.

Each variable x_i and $1 \leq i \leq n$ in an MODD is associated with one or more nodes, which appear at the same *level* in the MODD. More precisely, the nodes associated with variable x_i corresponds to level-$(i-1)$ and vice versa. Therefore level-i, corresponding to variable x_{i+1}, can contain at most N^i nodes. Hence the root of the MODD contains exactly 1 node, and the level before the external nodes can contain at most N^{n-2} nodes.

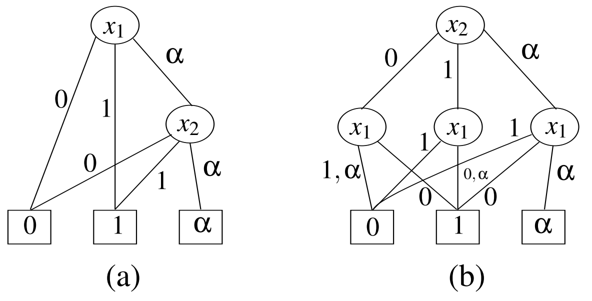


Figure 24: Effect of variable ordering.

Example 1 *Let us consider the MODD shown in Fig. 24(a). This MODD represents the following function in GF*(3)*:* $f(x_1,x_2) = g_1(x_1) + g_2(x_1)g_1(x_2) + \alpha \cdot g_2(x_1)g_2(x_2)$*, where* $g_r(x_s) = 1 - [x_s - \delta(r)]^2$.

Here both the levels 0 and 1, corresponding to the variables x_1 *and* x_2 *respectively, contain exactly 1 node each.*

Lemma 2 *Theorem 5 results in a disjoint expression, i.e. the product-terms in Eq. (20) are mutually (pairwise) disjoint.*

Proof. In Eq. (20) $g_e(x_k) = 1$ iff $x_k = \delta(e)$. For all other values of x_k $g_e(x_k) = 0$. Let us consider any two literals $g_r(x_k)$ and $g_s(x_k)$ such that $r \neq s$. Two cases may arise:

Case I: $x_k \neq \delta(r)$ and $x_k \neq \delta(s)$. In this case both $g_r(x_k)$ and $g_s(x_k)$ will equate to 0. Therefore, $g_r(x_k) \cdot g_s(x_k) = 0$.

Case II: Either $x_k = \delta(r)$ or $x_k = \delta(s)$, but not both. If $x_k = \delta(r)$, then $g_r(x_k) = 1$ and $g_s(x_k) = 0$; otherwise, $g_r(x_k) = 0$ and $g_s(x_k) = 1$. Therefore, $g_r(x_k) \cdot g_s(x_k) = 0$.

Hence the proof follows. [Q.E.D.]

Lemma 2 yields the following.

Theorem 6 *Each path from the root node to a non-zero terminal node in an MODD represents a disjoint product term in GF*(N).

Example 2 *Let us consider the MODD in Fig. 24(a) representing a function in GF*(3)*. The path* $\alpha, 1$ *represents the product-term* $X = g_\alpha(x_1)g_1(x_2)$*. The path* α, α *represents the product-term* $Y = g_\alpha(x_1)g_\alpha(x_2)$*. Clearly X and Y are disjoint, because* $X \cdot Y = 0$ *as* $g_1(x_2) \cdot g_\alpha(x_2) = 0$.

5.4 Reduction

We have the following from Theorem 5.

Corollary 6.1 *In Eq. (20), if $\forall i, j \in I_N$ and $i \neq j$ $f|_{x_k=\delta(i)} = f|_{x_k=\delta(j)}$, then $f = f|_{x_k=\delta(0)} = f|_{x_k=\delta(1)} = \cdots = f|_{x_k=\delta(N-1)}$.*

Proof. By perfect induction. Let $h = f|_{x_k=\delta(0)} = f|_{x_k=\delta(1)} = \cdots = f|_{x_k=\delta(N-1)}$. Then Eq. (20) becomes, $f = h\sum_{e=0}^{N-1} g_e(x_k)$.

For any $a \in \mathrm{GF}(N)$ such that $a \neq 0$, $a^{N-1} = 1$. Now in Theorem 5 if $x_k = \delta(r)$, then $g_r(x_k) = 1$ for $r \in I_N$. Furthermore, $g_s(x_k) = 0$, $\forall s \in I_N$ and $s \neq r$. Therefore, $\sum_{e=0}^{N-1} g_e(x_k)$ becomes 1, which implies $f = h$. Hence the proof. [Q.E.D.]

Based on the above, an MODD can be reduced as outlined in the following.

Reduction Rules There are two reduction rules.

- If all the N children of a node v point to the same node w, then delete v and connect the incoming edge of v to w. This follows from Corollary 6.1.
- Share equivalent subgraphs.

A DD in GF(N) is said to be ordered if the expansion in Eq. (20) is recursively carried out in a certain linear variable order such that on all the paths throughout the graph the variables also respect the same linear order.

A DD in GF(N) is said to be reduced iff: (a) There is no node $u \in V$ such that $\forall i, j \in I_N$ and $i \neq j$, $\mathrm{child}_i(u) = \mathrm{child}_j(u)$. ($b$) There are no two distinct nodes $u, v \in V$ which have the same variable names and same children, i.e. $var(u) = var(v)$ and $\mathrm{child}_i(u) = \mathrm{child}_i(v)$ $\forall i \in I_N$ implies $u = v$.

We have the following from the definition of the MODD.

Lemma 3 *For any node v in a reduced DD in GF(N), the subgraph rooted at v is itself reduced.*

Canonicity A reduced ordered MODD in GF(N) based on the expansion of Theorem 5 is canonical up to isomorphism. This is stated in the following.

Theorem 7 *For any n variable function $f(x_1, x_2, \ldots, x_n)$ in GF(N) there is exactly one reduced ordered DD in GF(N) with respect to a specific variable order, which is minimal with respect to the reduction rules.*

Proof. The proof is done by induction on the number of arguments n in f.

Base Case: If $n = 0$, then the function yields a constant in GF(N). The resulting reduced ordered DD in GF(N) has exactly one node, namely the terminal node with a value in GF(N). Therefore, any function in GF(N) with $n = 0$ will have exactly one external node with the same value in GF(N), and hence will be unique. Note that any reduced ordered DD in GF(N) with at least one non-terminal node would yield a non-constant function.

Induction Hypothesis: Assume that the theorem holds for all functions with $n-1$ arguments.

Induction Step: We show that the theorem holds for any function f in GF(N) with n arguments. Without loss of generality, let us assume that the variables are ordered as $(x_1, x_2, \ldots, x_n)$. Considering the first variable x_1,

$$f(x_1, x_2, \ldots, x_n) = \sum_{e=0}^{N-1} g_e(x_1) f|_{x_1=\delta(e)}, \tag{21}$$

where $g_e(x_1) = 1 - [x_1 - \delta(e)]^{N-1}$.

Let f^z represent the function realized by the subgraph rooted by z. In a reduced ordered DD in GF(N) for f, each $f|_{x_1=\delta(i)}$, $\forall i \in I_N$, is represented by a subgraph rooted by u_i. Since, each $f|_{x_1=\delta(i)}$ is a function of $n-1$ variables, so each u_i, $\forall i \in I_N$, represents a unique reduced ordered DD in GF(N), by the induction hypothesis.

Two cases may arise:

Case 1: $\forall i, j \in I_N$ and $i \neq j$, $u_i = u_j$. It must be the case that, $f^{u_i} = f^{u_j}$, which implies $f|_{x_1=\delta(i)} = f^{u_i} = f^{u_j} = f|_{x_1=\delta(j)}$. Therefore, $u_i = u_j$, $\forall i, j \in I_N$ and $i \neq j$, is a reduced ordered DD in GF(N) for f. This is also unique, since if its not then owing to the ordering x_1 would appear in the root node v if it appears at all. This would imply that $f = f^v$. Therefore, from the definition of a DD in GF(N), we must have $f|_{x_1=\delta(k)} = f^v|_{x_1=\delta(k)} = f^{child_k(v)}$, $\forall k \in I_N$. Since, by assumption, $f|_{x_1=\delta(i)} = f^{u_i} = f^{u_j} = f|_{x_1=\delta(j)}$, $\forall i, j \in I_N$ and $i \neq j$, this would imply that all the children of v would be the same. This is a contradiction, because it violates the fact that the DD is reduced. Hence, the reduced ordered DD in GF(N) must be unique.

Case 2: $\exists S \subseteq I_N$ such that $\forall k, l \in S$ and $k \neq l$, $u_k \neq u_l$. Therefore, by the induction hypothesis, $f^{u_k} \neq f^{u_l}$, $\forall k, l \in S$ and $k \neq l$. Let w be a node with $var(w) = x_1$, and $child_q(w) = u_q$, $\forall q \in I_N$.

Therefore,

$$f^w = \sum_{e=0}^{N-1} g_e(x_1) f^{u_e},$$

where $g_e(x_1) = 1 - [x_1 - \delta(e)]^{N-1}$, and the DD in GF($N$) rooted by w is reduced.

By assumption $f^{u_t} = f|_{x_1=\delta(t)}$, $\forall t \in I_N$. Therefore, from Eq. (21), $f^w = f$.

Suppose that this reduced ordered DD in GF(N) is not unique, and that there exists another reduced ordered DD in GF(N) for f rooted by w'. Now, $f^{w'} = f$, and it must be the case that $f^{w'}|_{x_1=\delta(k)} \neq f^{w'}|_{x_1=\delta(l)}$, $\forall k, l \in S$ and $k \neq l$. If this is not the case, then $f|_{x_1=\delta(i)} = f^{w'}|_{x_1=\delta(i)} = f^{child_i(w')} = f^{child_j(w')} = f^{w'}|_{x_1=\delta(j)} = f|_{x_1=\delta(j)}$, $\forall i, j \in I_N$ and $i \neq j$. This is a contradiction, since this would imply that all the children of w' would be the same, thus violating the fact that the DD is reduced.

Now, due to the ordering, $var(w') = x_1 = var(w)$. In addition, since $f^{w'} = f$, it follows that $f^{child_r(w')} = f|_{x_1=\delta(r)} = f^{u_r} = f^{child_r(w)}$, $\forall r \in I_N$. Therefore, by the induction hypothesis, $child_r(w') = u_r = child_r(w)$, $\forall r \in I_N$. Therefore, it follows that $w = w'$, by induction. Hence the proof for uniqueness follows by induction.

Minimality We now prove the minimality of a reduced ordered DD in GF(N) in terms of the total number of nodes, with respect to the reduction rules. Suppose that the reduced ordered DD in GF(N) for f is not minimal with respect to the reduction rules. Then we can find a smaller DD in GF(N) for f as follows. If the DD in GF(N) contains a node v with $child_i(v) = child_j(v)$, $\forall i, j \in I_N$ and $i \neq j$, then eliminate v and for any node w with $child_k(w) = v$ $(k \in I_N)$, make $child_k(w) = child_0(v)$.

If the DD in GF(N) contains distinct but isomorphic subgraphs rooted by v and v', then eliminate v' and for any node w such that $child_k(w) = v'$ $(k \in I_N)$, make $child_k(w) = v$. [Q.E.D.]

A Reduction Algorithm A reduction algorithm for MODD appears in Fig. 25. In order for sharing of equivalent subgraphs, subgraphs already present in the MODD are placed in a table. In Lines 4, 6, 12, and 14 we have assumed that checking for membership and addition of an element to such a table (HT) can be carried out in constant times, e.g. by using a hash-table. Hence, the complexity of the algorithm is $O(|G|)$ since each node can be made to be visited just once during the reduction process, where G is an MODD of a function before the reduction.

```
 1  Algorithm ReduceDD($v$ : node) : node;
 2  begin
 3      if $v$ is a terminal node, then
 4          if IsIn($v$,$HT$), then return LookUp($HT$,$v$)
 5          else begin
 6              $HT$ := insert($v$,$HT$); (* Initially $HT = \varnothing$ *)
 7              return $v$
 8          end;
 9      else begin
10          $v'_i$ := ReduceDD(child$_i(v)$), $\forall i \in I_N$;
11          if $v'_j = v'_k$, $\forall j,k \in I_N$ and $j \neq k$, then return $v'_0$
12          else if IsIn($v$,$HT$), then return LookUp($HT$,$v$)
13          else begin
14              $HT$ := insert($v$,$HT$);
15              return $v$
16          end
17      end
18  end;
```

Figure 25: A reduction algorithm for MODDs in GF(N).

However, an algorithm for the creation of MODDs from the functional description in GF(N) will have a complexity $O(N^n)$ in the worst case. The algorithm for the creation of MODDs can be derived from Eq. (20). The efficiency of the algorithm can be improved in general by noting that during the recursive expansion with respect to each variable, certain variables may not appear for further recursive calls. Therefore the recursion tree need not be expanded in the direction of a variable which does not appear. The efficiency can be further improved by incorporating Lemma 4 based on a dynamic programming like approach as discussed in Section 5.6, which has been done for the experimental results in Section 5.11. In this case the network in GF(N) is traversed in the topological order from the inputs to the outputs and Lemma 4 is applied iteratively.

Note that the size of a reduced MODD depends heavily on the variable ordering, as in any other DD [1, 82]. The depth of an MODD in is $O(n)$ in the worst case since each variable appears once at each level in the worst case.

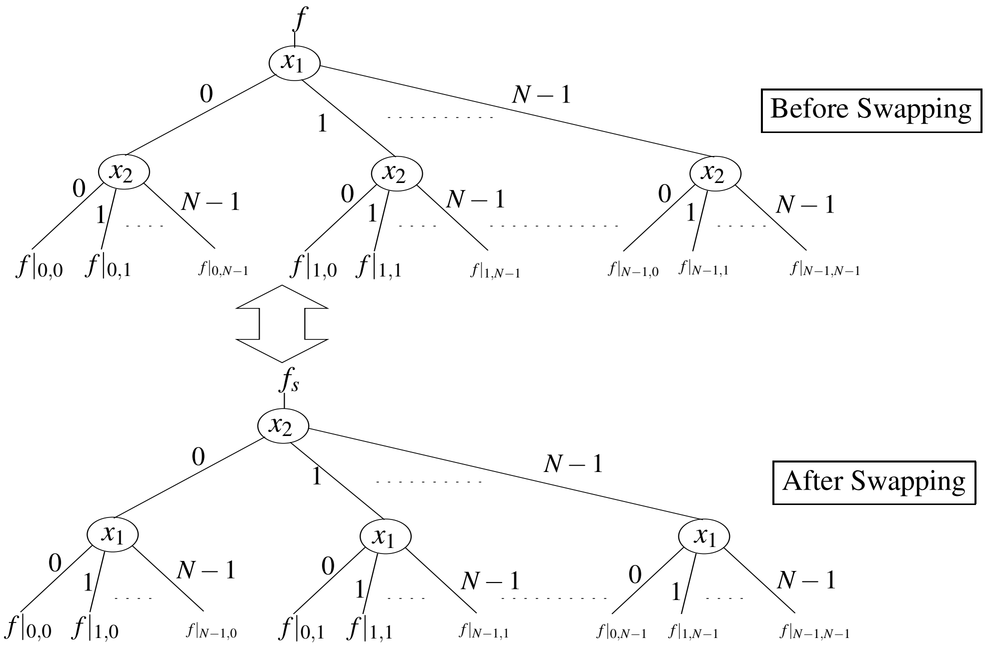

f
x_1
0
1
$N-1$
Before Swapping
x_2
$f|_{0,0}$ $f|_{0,1}$ $f|_{0,N-1}$ $f|_{1,0}$ $f|_{1,1}$ $f|_{1,N-1}$ $f|_{N-1,0}$ $f|_{N-1,1}$ $f|_{N-1,N-1}$
f_s
After Swapping
$f|_{0,0}$ $f|_{1,0}$ $f|_{N-1,0}$ $f|_{0,1}$ $f|_{1,1}$ $f|_{N-1,1}$ $f|_{0,N-1}$ $f|_{1,N-1}$ $f|_{N-1,N-1}$

Figure 26: Theory behind reordering.

5.5 Variable Reordering

The size, i.e. number of nodes, of an MODD depends on the order of the variables during its construction. For example, the MODD in Fig. 24(a) represents a function in GF(3) under the variable order (x_1,x_2). Fig. 24(b) shows the same function in GF(3), but under the variable order (x_2,x_1). Clearly Fig. 24(a) contains fewer nodes than Fig. 24(b). Given an n variable function in GF(N), the size of the solution space for finding the best variable order is $O(n!)$, which is impractical for large values of n. Hence, a heuristic level-by-level swap based algorithm is considered in this chapter.

The theory behind variable reordering in GF(N) is based on Theorem 5 as follows. Without loss of generality let us assume that variables x_1 and x_2 are to be swapped. From Theorem 5 we have,

$$\begin{aligned} f(x_1,x_2,\ldots,x_n) &= g_0(x_1)\big(g_0(x_2)f|_{0,0}+g_1(x_2)f|_{0,1}+\cdots+g_{N-1}(x_2)f|_{0,N-1}\big)\\ &+ g_1(x_1)\big(g_0(x_2)f|_{1,0}+g_1(x_2)f|_{1,1}+\cdots+g_{N-1}(x_2)f|_{1,N-1}\big)+\cdots\\ &+ g_{N-1}(x_1)\big(g_0(x_2)f|_{N-1,0}+g_1(x_2)f|_{N-1,1}+\cdots+g_{N-1}(x_2)f|_{N-1,N-1}\big). \end{aligned}$$

If x_1 and x_2 are swapped, then again from Theorem 5 the following function f_s results,

$$\begin{aligned} f_s(x_1,x_2,\ldots,x_n) &= g_0(x_2)\big(g_0(x_1)f|_{0,0}+g_1(x_1)f|_{1,0}+\cdots+g_{N-1}(x_1)f|_{N-1,0}\big)\\ &+ g_1(x_2)\big(g_0(x_1)f|_{0,1}+g_1(x_1)f|_{1,1}+\cdots+g_{N-1}(x_1)f|_{N-1,1}\big)+\cdots\\ &+ g_{N-1}(x_2)\big(g_0(x_1)f|_{0,N-1}+g_1(x_1)f|_{1,N-1}+\cdots+g_{N-1}(x_1)f|_{N-1,N-1}\big), \end{aligned}$$

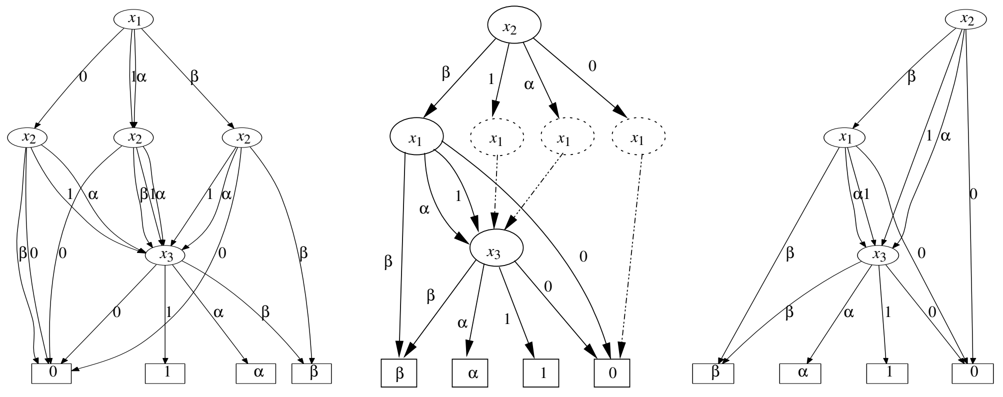
where $f \equiv f_s$. It can be noted by comparing f and f_s that in f_s each literal $g_e(x_1)$ ($g_e(x_2)$) is swapped with $g_e(x_2)$ ($g_e(x_1)$) for $e \in I_N$, and each cofactor $f|_{r,s}$ ($f|_{s,r}$) is swapped with $f|_{s,r}$ ($f|_{r,s}$). Fig. 26 shows the MODDs corresponding to f (top MODD) and f_s (bottom MODD). In the top MODD variables x_1 and x_2 appear in levels 0 and 1 respectively, while the cofactors appear as external nodes. This can be a more general case, e.g. the two variables may appear in any arbitrary but consecutive levels in a larger MODD, e.g. variables x_i and x_{i+1} and $2 < i \leq n$. Clearly to swap two variables all we have to do is: (i) swap the contents of the nodes (i.e. the variables) in the two levels, and (ii) swap each cofactor $f|_{r,s}$ ($f|_{s,r}$) with $f|_{s,r}$ ($f|_{r,s}$). However, care must be exercised when a level has one or more missing nodes due to reduction. If this happens, then the missing nodes may have to be recreated in the swapped version. Also, some nodes may become redundant after the swap, in which case the redundant nodes must not appear in the final result (refer to Example 3). However, if a node at level-i ($0 \leq i < n$), which is to be swapped with level-$i+1$, does not have any children at level-$i+1$, then it can be moved to level-$i+1$ directly. By similar reasoning, if a node at level-$i+1$ does not have any parent nodes at level-i, then that node can be moved up to level-i directly. Note that swapping two levels i and $i+1$ does not affect the other levels, i.e. those in the range $0 \leq j < i$ (if $i > 1$) and $i+1 < k < n$ (if $i < n-1$).

The heuristic swap based variable reordering algorithm presented in the following is based on this (Theorem 5). A swap based variable reordering algorithm exists for BDD [8], but it is not suitable for MODD reordering.

The algorithm uses an array of hash-tables, where the array indexes correspond to the levels in an MODD for direct access to each of the nodes within a level. It proceeds by sifting a selected level (i.e. a variable) up or down by swapping it with a previous or next level. The level with the largest number of nodes is considered first, and then the one with the next largest node count, etc., i.e. the array of hash-tables is sorted in descending order of the hash-table sizes. Once the level to be sifted first is considered it is sifted up if it is closer to the root, or down if it is closer to the external nodes. If it lies in the middle, then the decision to either sift up or down is made arbitrarily. The algorithm stops after a complete sift-up and down operations. The complexity of the algorithm can be argued to be $O(n^2)$ [8]. Various heuristics have been considered to limit the sift and swap operations for speed up. For example, if a sift-up (down) operation doubles the node count, no more sift-up (down) operations are carried out.

Example 3 *Fig. 27 shows the basic idea behind the swap algorithm. Fig. 27(a) shows the original MODD for a function $f(x_1, x_2, x_3)$ in $GF(4)$ generated by recursive expansion of Theorem 5. Here, variables x_1, x_2 and x_3 appear in levels 0, 1, and 2 respectively. Level 1 (i.e. variable x_2) contains the largest number of nodes. Hence, this is considered to be the starting point of the sift operation. Level 1 is equidistant from level-0 and level-2. Hence, a sift-up is chosen arbitrarily. Swap between level-0 and 1 results in Fig. 27(b). The nodes shown with broken lines are redundant owing to the fact that all their children point to the same node (Corollary 6.1). Hence, these nodes are not considered in the final result of Fig. 27(c). For example, considering the paths with the edge $x_2 = 1$ in Fig. 27(b), all the paths with edge $x_2 = 1$ leading to node x_3 in Fig. 27(a) have the edges with variable $x_1 = i$ for $i = 0, 1, \alpha, \beta$. Therefore node x_1 becomes redundant by Corollary 6.1, if node x_1 appears after node x_2 under this circumstance. The same reasoning applies to the other two nodes with variable x_1 shown with the broken lines.*

It can be shown that further sift operations do not yield additional node reduction. The original



(a) Before Reordering (b) Intermediate Stage (c) After Reordering

Figure 27: Reordering example.

node count was 5, and the new node count is 3.

5.6 Operations in GF(N)

Algebraic Operations Algebraic operations such as addition, multiplication, subtraction, and division in GF(N) can be carried out between two MODDs. Consider the following lemma, which can be shown to hold by perfect induction.

Lemma 4 *Let $f(x_1,\ldots,x_i,\ldots,x_n)$ and $h(x_1,\ldots,x_i,\ldots,x_n)$ be two functions in GF(N), and '$\odot$' represent an algebraic operation in GF(N). Then*

$$f \odot h = \sum_{e=0}^{N-1} g_e(x_i)(f|_{x_i=\delta(e)} \odot h|_{x_i=\delta(e)}), \tag{22}$$

where $g_e(x_i) = 1 - [x_i - \delta(e)]^{N-1}$.

Lemma 4 can be implemented recursively to perform algebraic operations between MODDs. However, application of Lemma 4 directly will almost certainly be explosive in terms of the search space. Two things can be done to eliminate this. Firstly, while the resulting DD in GF(N) is being constructed, it can be reduced at the same time. Secondly, intermediate results can be stored in a cache (dynamic programming), thus eliminating many operations, which will otherwise have to be repeated.

Let G_f and G_h be the reduced MODDs for f and h respectively. The complexity of such an operation can be reasoned about by considering the case for BDDs [1]. Assuming that insertion and deletion from the cache can be carried out in constant times, owing to the dynamic programming nature the number of recursive calls can be limited to $O(|G_f| \cdot |G_h|)$.

Composition We have the following, which can be shown to hold by perfect induction.

Lemma 5 *Let $f(x_1,x_2,\ldots,x_i,\ldots,x_n)$ and $h(x_1,x_2,\ldots,x_n)$ be two functions in GF(N). Then,*

$$f|_{x_i=h} = \sum_{e=0}^{N-1} [1-(h-\delta(e))^{N-1}] f|_{x_i=\delta(e)}. \tag{23}$$

An algorithm for composition of two functions in GF(N) can be formed based on Lemma 5 in a manner similar to that for a BDD [1]. For this operation we require a *restrict operation* in GF(N), similar to that for a BDD, and the algebraic operations presented previously. The restrict algorithm can be constructed for a reduced ordered DD in GF(N) in a manner similar to that for a BDD, and is not shown here for brevity.

Multiple-Valued SAT Given a function $f(x_1,x_2,\ldots,x_n)$ in GF(N) and $T \subseteq I_N - \{0\}$, the idea is to find an assignment for x_i, $\forall i \in \{1,2,\ldots,n\}$, such that the value of f is in $\{\delta(s)|s \in T\}$. If such an assignment exists, then f is said to be satisfiable (MV-SAT); otherwise it is unsatisfiable.

The MV-SAT problem finds applications in bounded model checking, simulation, testing and verification. An algorithm for *any* such satisfying assignment will have a complexity $O(|G_f|)$, where G_f is the reduced MODD for the function f in GF(N). An algorithm for *all* such assignments would have an exponential complexity. However, this process can be speeded up by considering characteristic and encoded characteristic functions in GF(N) and their evaluation times as discussed in Sections 5.9 and 5.10 [93, 94].

5.7 Multiple-Output Functions in GF(N)

For multiple input multiple output binary functions, the inputs or outputs can be arbitrarily grouped into m-bit chunks and each m-bit chunk can be represented in GF(2^m) with a single MODD. Further node reduction can be obtained by sharing the nodes between each of the MODDs representing a chunk of bits. Such an MODD will be called a *shared MODD* or SMODD. The general idea is shown in Fig. 5.7. The SMODD is basically a single diagram with multiple root nodes, which is also canonic. The canonicity of the SMODD can be argued in a similar manner as in a single MODD.

Similar reasoning can be carried over to higher order fields. Given any multiple output function in GF(R), where R is a power of a prime, the inputs and outputs can be arbitrarily grouped into m R-valued chunks and each chunk can be represented in GF(R^m) by means of an MODD. Then an SMODD will represent all the chunks simultaneously.

The concept of levels are applicable to SMODDs across all the outputs simultaneously by trivial reasoning. Therefore, the theory behind variable reordering, as discussed in Section 5.5, applies equally well to SMODDs. In this case when levels i and $i+1$ $(0 \leq i < n)$ are swapped, all the nodes in levels i and $i+1$ across all the outputs have to be considered simultaneously. Therefore, the swap based sift reordering algorithm discussed in Section 5.5 works equally well for SMODDs as it does for MODDs.

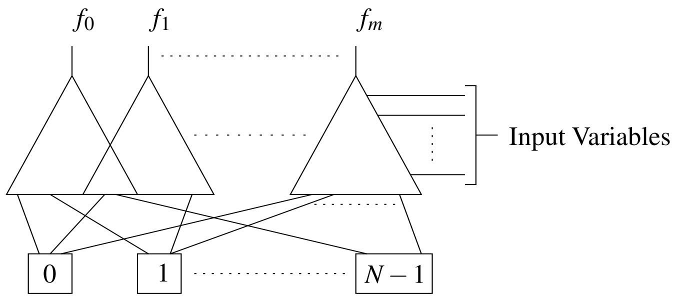


Figure 28: General Structure of Shared MODD.

5.8 Further Node Reduction

Further node reduction can be obtained by means of the following two rules, in addition to the two rules presented in Section 5.3.

- *Zero Suppression:* Suppress the 0-valued terminal node, along with all the edges pointing to it.
- *Normalization:* Move the values of the non-zero terminal nodes as weights to the edges, and ensure that (i) the weight of a specific valued edge (e.g. that with the highest value) is always 1, and (ii) assuming P represents the set of all the paths, $\forall z \in P$ the GF(N) product of all the weights along z is equal to the value of the function corresponding to z.

Note that the zero suppression rule is unlike the reduction rule for the zero suppressed BDD [97]. It can be argued that the above two rules also will maintain the canonicity, if the weights are assigned in a fixed order throughout the graph during normalization. A reduced graph obtained using the above four reduction rules in GF(N) will be called a *(Z)ero-suppressed (N)ormalized MODD* or an ZNMODD. The values of the terminal nodes in an MODD is *distributed* as weights over each path in the ZNMODD. To read off a value of a function from a ZNMODD, first the path corresponding to the inputs is determined. Then all the weights along that path is multiplied in GF(N), which corresponds to the value of that function. In the rest of the chapter the weight of the highest valued edge will be normalized to 1, unless otherwise stated.

Example 4 *Let us consider the function* $f(x_1,x_2) = [0\beta1001\alpha000000000]$ *in GF*(4)*, where* $\{0,1,\alpha,\beta\}$ *are the elements of GF*(4)*. Fig. 29(a) shows this function realized by means of a reduced MODD. Fig. 29(b)–Fig. 29(d) shows the gradual conversion to ZNMODD. Here, the lines with zero, one, two, and three cuts represent the values 0, 1,* α*, and* β *respectively. Note how the weights are moved around and adjusted.*

In Fig. 29(b) the terminal node with 0-value is suppressed along with all the edges pointing to it. Also the non-zero values of the terminal nodes are moved as weights associated with the terminal edges. Let us normalize with respect to the highest valued edge, i.e. make the weight of the highest valued edges, β *in this case, equal to 1. The* α*-edge of the left sub-graph rooted at* x_2 *has a weight of* α*. Therefore, in order to make its weight equal to 1,* α *is moved up one level, while the 1-edge*

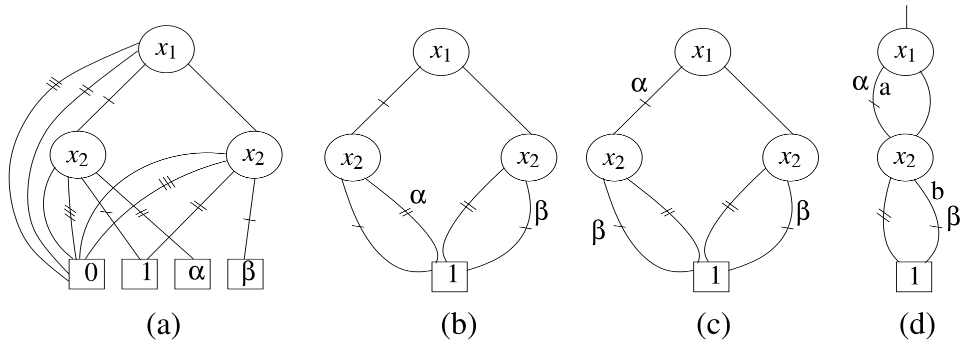


Figure 29: Example of ZNMODD reduction.

is assigned a weight of β *to maintain correctness of the underlying function. This results in the ZNMODD of Fig. 29(c). Clearly in Fig. 29(c) the two sub-graphs rooted at* x_2 *are isomorphic, which can be shared resulting in the ZNMODD of Fig. 29(d).*

Now let us find the value of $f(1,1)$. *This should yield a 1. From Fig. 29(d), this corresponds to the path ab. The value of this function is therefore* $1 \cdot \alpha \cdot \beta = \alpha \cdot \beta = 1$. *Similarly,* $f(1,\alpha)$ *yields* $1 \cdot \alpha \cdot 1 = \alpha$*, and so on.*

Note that the total number of paths in Fig. 29(a) is 10, while that in Fig. 29(d) is only 4.

5.9 Representing Characteristic Functions in GF(N)

The characteristic function (CF) defines a relation over inputs and outputs such that CF $= 1$ if for a specific input combination the output is valid; otherwise CF $= 0$.

Let us consider a multiple output function defined over finite fields: $f(x_1, x_2, \ldots, x_n) = (y_1, y_2, \ldots, y_m)$. Let $X = (x_1, x_2, \ldots, x_n)$ and $Y = (y_1, y_2, \ldots, y_m)$. Then the $(n+m)$-input 1-output CF is defined as

$$\phi(X,Y) = \begin{cases} 1 & \text{if } f(X) = Y \\ 0 & \text{otherwise} \end{cases}$$

An SMODD can be constructed from the above which will constitute the n input variables and m *auxiliary* variables (AV) corresponding to each of the outputs. Such an SMODD will be called CF-SMODD. Given an input combination of f, the nodes corresponding to the AVs in the CF-SMODD decide the outputs of f. For each node corresponding to an AV, except for only one edge, all the other edges lead to the terminal node 0. The edge leading to the non-zero terminal node determines the output of the function. Examples of the CF can be found in [98].

The concept of CF can be extend by allowing output encoding, since there is only one possible set of outputs for a given input combination. The resulting function can be represented by a mapping $\eta : G^n \times G^l \to G$, where $l = \lceil \log_N(m) \rceil$ and will be called *encoded* CF or ECF. The ECF has l AVs. Each output is defined by one of the N^l input combinations in an l AV function. As with the CF-SMODD, an ECF can be represented by means of an SMODD, which we shall call the ECF-SMODD. The following example illustrates the key points.

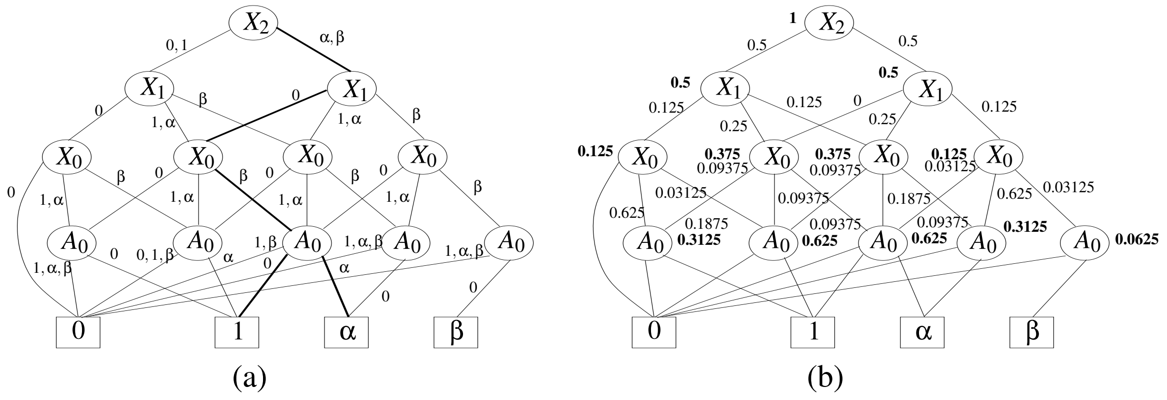


Figure 30: (a) ECF-SMODD, (b) Computation of APL.

Example 5 *Let us consider a 5-input 3-output binary function defined as follows, with the inputs denoted by the variables* $(x_0, x_1, x_2, x_3, x_4)$ *and the outputs denoted by* (f_0, f_1, f_2).

$$\begin{aligned} f_0 &= \sum m(15,23,26,29,30,31) \\ f_1 &= \sum m(1,2,4,8,11,13,14,16,21,22,26,31) \\ f_2 &= \sum m(3,5,6,7,8,9,14,11,12,13,17,19,20,21,22,23,24,25,26,28) \end{aligned}$$

Let us assuming that the function is encoded in $GF(4)$ *with inputs and outputs grouped as* $X_0 = (x_0, x_1)$, $X_1 = (x_2, x_3)$, $X_2 = x_4$, $F_0 = (f_0, f_1)$, $F_1 = f_2$. *The CF of this function is* $\phi(X_0, X_1, X_2, F_0, F_1)$. *We have assumed that the binary combinations* $10 = \alpha$ *and* $11 = \beta$. *The variables* F_0 *and* F_1 *can be encoded as* $A_0 = 0$ *for* F_0 *and* $A_0 = \alpha$ *for* F_1, *resulting in the ECF* $\eta(X_0, X_1, X_2, A_0)$. *Note that we can encode 4 functions using one AV. Here we have only two functions. The resulting ECF-SMODD appears in Fig. 30(a).*

5.10 Evaluation of Functions

The evaluation of the SMODDs is required for finding satisfying assignment corresponding to an input pattern. The path corresponding to the given input pattern is traced from the root node to one of the terminal nodes, and the value of the terminal node gives the satisfying assignment, if it exists. This is an $O(n)$ operation, which can become a bottleneck especially when the number of inputs is large and there are many input patterns require evaluating. However, fast evaluation is highly desirable for applications in simulation, testing, and safety checking [93, 94].

In the case of a CF the outputs are evaluated at the AVs. As we know, all the outgoing edges except only one edge lead to the zero terminal node. The remaining edge indicates the value of the function. With the ECF once we reach the node corresponding to the AV, the paths corresponding to each of the output encoding is traced to find the value.

For example, lets consider an input pattern $(x_0 = 1, x_1 = 1, x_2 = 0, x_3 = 0, x_4 = 1)$ for the ECF-SMODD of Fig. 30. The pattern is $(X_2 = \alpha, X_1 = 0, X_0 = \beta)$ if it is encoded in GF(4) (Example 5). The path traced by the evaluation is shown in bold. After reaching node A_0 in the path, the path corresponding to the given encoding for each output has to be taken into account. In this case the encoding was defined as $A_0 = 0$ for F_0 and $A_0 = \alpha$ for F_1. Hence, if we take the path corresponding to $A_0 = 0$ we end up at terminal node 1, thus giving us $F_0 = 1$. Similarly, $F_1 = \alpha$. This results in $(f_0 = 0, f_1 = 1, f_2 = 1)$, which is the required evaluation.

Comparison of Evaluation Times Functions can be represented and evaluated by means of SMODD, CF-SMODD, or ECF-SMODD. This section provides a mechanism for comparing the evaluation times for each of the cases. We have tested our theory on many benchmark, and the results appear in Section 5.11. A good estimation of evaluation times can be obtained by computing the *average path length* (APL), which we define below.

1. *Node Traversing Probability ($P(V_i)$):* It is the probability of traversing the node V_i when an MODD is traversed from the root node to a terminal node.

2. *Edge Traversing Probability ($P(e_{j,v_i})$):* It is the probability of traversing the edge $j = 0, 1, \ldots, p^{m-1}$ from the node V_i, i.e. $P(e_{j,v_i}) = \frac{P(V_i)}{p^m}$, for nodes corresponding to the input variables.

3. The edge traversing probability for edges emanating from a node corresponding to an AV is $P(e_{j,v_i}) = P(V_i)$ since all the edges have to be traversed for determining all the outputs of the function.

4. The node traversing probability is equal to the sum of all the edge traversing probabilities incident on it.

5. *Average Path Length (APL):* For an SMODD the APL is equal to the sum of the node traversing probabilities of the non terminal nodes. For a CF-SMODD and ECF-SMODD the APL is equal to the sum of the node traversing probabilities of those nodes above the AVs, and the APLs for each subgraph rooted at the AVs.

6. The average path length of a shared MODD is the sum of the average path lengths of the individual MODDs.

An algorithm for computing the APL for ECF-SMODD appears in Fig. 31. Algorithms for computing the APLs for SMODD and CF-SMODD can be formulated from this algorithm, which we have implemented in Section 5.11, but the details have been left out for brevity.

For example the node and edge traversing probabilities of the ECF-SMODD in Fig. 30(a) appears in Fig. 30(b). The first three levels in the tree correspond to the input variables. Hence the probabilities are computed using definitions 1, 2 and 4. However, since all the outputs have to be considered, the APL for each subtree is separately computed at the auxiliary nodes. The probabilities at the auxiliary nodes correspond to the sums of the APLs of each sub-tree of the outputs. The APL is $1 + (0.5 + 0.5) + (0.125 + 0.375 + 0.375 + 0.125) + (0.3125 + 0.625 + 0.625 + 0.3125 + 0.0625) = 4.9375$.

```
Algorithm ENC_PROB(int $NODE$, float $PROB$)
{   // $NODE$ is the current node, and $PROB$ is its probability
    if ($NODE =$ Terminal Node) return 0;
    Add $PROB$ to $NODE$'s probability; // Each node stores its probability
    Decrease the Reference count of $NODE$ by 1;
    if (Reference count of $NODE > 0$) return 0;
    float $TOT = 0$;
    for(int $i = 0; i < N_o; i++$){ // $N_o$ is the output field size
        $T = \text{child}_i(NODE)$;
        if($NODE =$ auxiliary node){
            clear reference counts for all nodes of the subtree $T$;
            make new reference count only for $T$;
            clear probability variable in all nodes of T;
            APL for $T =$ ENC_PROB($T$, $NODE$'s probability);
            $TOT = TOT +$ APL for $T + NODE$'s probability;
        }else{
            APL for $T =$ ENC_PROB($T$, $NODE$'s probability$/N_o$);
            $TOT = TOT +$ APL for $T$;
        }
    }
    if($NODE != $ auxiliary node) $TOT = TOT + NODE$'s probability;
    return $TOT$;
}
```

Figure 31: Algorithm for calculation of APL in ECF-SMODD.

5.11 Experimental Results

The techniques in this chapter have been applied to a number of benchmarks, including integer multiplier circuits. The program was developed in C++ (Gnu C++ 3.2.2) and tested on a Pentium-4 machine with 256MB RAM, running RedHat Linux 9 (kernel 2.4).

Performance Table 6 shows results from the standard IWLS'93 and MCNC benchmark sets. Columns 'I/P' and 'O/P' represent the total number of inputs and outputs, column 'SBDD' shows the number of nodes obtained using the shared ROBDD representation, while column GF(2^r) represents the number of nodes obtained based on the proposed DD for a field size of 2^r. In column GF(2^r) r adjacent bits are grouped together for each variable in GF(2^r). The nodes of the proposed DD are also shared across the outputs. The same notation is used for the other tables.

The columns with the headings "W/o reord" and "Reord" present the node count without and with variable reordering. A first-come first-served variable ordering is considered for the "W/o reord" columns. For the reordering the variable-by-variable swap based sifting algorithm of Section 5.5 has been employed. Significant node reduction is apparent for many circuits, e.g. misex3c, etc. However, in some cases the node count has increased owing to the lack of sharing. Note that successive swap operation in a particular direction (up or down) is only carried out if a swap operation does not increase the size of the MODD by two or more. This restriction can be relaxed (e.g. by considering the maximum allowable size increase to be 1.5 times) to obtain better results sometimes. However, this also increases the execution time, as more swaps are carried out. Variable

Table 6: Same Field Size for input and output.

Benchmark	I/P	O/P	SBDD	GF(2^2)		GF(2^3)		GF(2^4)	
				W/o reord	Reord	W/o reord	Reord	W/o reord	Reord
5xp1	7	10	88	48	42	43	35	27	16
9sym	9	1	33	17	17	10	10	-	-
apex4	9	19	1021	536	515	324	324	241	136
b12	15	9	91	78	47	48	45	61	51
bw	5	28	118	76	72	59	47	59	21
clip	9	5	254	136	89	88	41	49	31
misex3c	14	14	844	505	279	396	181	586	156
duke2	22	29	366	793	507	783	445	601	453
table5	17	15	685	751	678	636	636	499	348
e64	65	65	194	943	569	858	601	624	495
cordic	23	2	75	29	28	21	20	15	15
misex2	25	18	100	93	81	50	42	45	41
pdc	16	40	596	433	310	384	384	212	212
spla	16	46	628	352	339	261	261	155	155
ex1010	10	10	1402	662	654	346	344	670	207

reordering algorithms exist for MDDs [99]. However, they cannot be directly compared with the presented technique because the presented technique has been applied to varying input and output field sizes, while [99] seems to have ignored this aspect.

Apart from the benchmark 9sym, all the circuits have been tested up to GF(16). 9sym is a single output circuit and hence its testing in higher order fields seemed to be unjustified. In Table 6, the input and output field sizes are kept the same. Clearly as we move to a higher order field the number of nodes is reducing for the majority of the cases. Also, as we move to a higher order field, node reduction owing to reordering seems to be more effective.

For the rest of the tables, apart from Table 5.11, reordering has not been done to illustrate the other properties of the MODD more effectively, e.g. the effect on the node count when the input and output field sizes are varied, and when the MODDs are ordered based on the evaluation times.

Table 7: I/P Field varying with constant output field size of 2.

Benchmark	I/P	O/P	SBDD	GF(2^2)	GF(2^3)	GF(2^4)
5xp1	7	10	88	57	51	43
9sym	9	1	33	18	16	22
apex4	9	19	1021	509	346	240
b12	15	9	91	69	56	60
bw	5	28	118	71	47	42
clip	9	5	254	149	106	69
misex3c	14	14	844	450	300	246

Table 7 represent the results for the same set of benchmarks under the same variable ordering but the input field size is varied while the output field size is kept at constant 2. Clearly the number of nodes have reduced further for many benchmarks as compared to Table 6. The reason seems to be improved sharing of nodes between the different outputs.

Table 8 shows the result with input field size kept at constant 2 and the output field size varied, again under the same variable ordering. In general the number of nodes seems to have increased

Table 8: O/P field size varying with constant Input field size of 2

Benchmark	I/P	O/P	SBDD	GF(2^2)	GF(2^3)	GF(2^4)
5xp1	7	10	88	87	77	87
9sym	9	1	33	-	-	-
apex4	9	19	1021	1031	989	983
b12	15	9	91	102	84	126
bw	5	28	118	154	148	140
clip	9	5	254	227	211	195
misex3c	14	14	844	903	978	1333

owing to lack of sharing between the different outputs. In other words, higher output field size seems to hamper sharing of nodes between different outputs even though the number of nodes in each output may reduce. This observation seems to be consistent with the conclusion drawn from Table 7.

Table 9: Same field size for both input and output.

Multiplier	SBDD	GF(2^2)		GF(2^3)		GF(2^4)	
		W/o reord	Reord	W/o reord	Reord	W/o reord	Reord
2*2	14	7	7	6	5	1	-
3*3	51	28	28	15	15	16	9
4*4	157	98	87	84	60	31	31
5*5	471	254	249	183	183	272	121
6*6	1348	795	731	736	624	431	428

Table 5.11 shows results for $n * n$ integer multipliers for $n = 2,3,4,5,6$. The input and output field sizes are kept the same in this table. Clearly a substantial reduction in the number of nodes is noticeable for the majority of the benchmarks. In some cases reordering has produced further improvement, e.g. the $4*4$ and $6*6$ multipliers in GF(8), etc. Although we have not explicitly shown the results in GF(2^5) and GF(2^6), MODD reported only 63 nodes as opposed to the 471 nodes for SBDD for the $5*5$ multiplier in GF(2^5). Also for the $6*6$ multiplier the number of nodes reported by the MODD in GF(2^6) is only 127 as opposed to 1348 for the SBDD, i.e. more than an order of magnitude reduction. Also this table suggests that the node reduction seems to improve as we are considering larger and more practical integer multipliers.

Table 10: I/P Field varying with constant output field size of 2.

Multiplier	SBDD	GF(2^2)	GF(2^3)	GF(2^4)
2*2	14	9	10	4
3*3	51	33	24	28
4*4	157	78	64	55
5*5	471	263	190	127
6*6	1348	695	389	465

Table 10 shows the results with varying input field size and a constant output field size of 2. Again, considerable improvement has been observed for some benchmarks owing to the improved sharing of the nodes across the outputs.

Table 11 shows the results for the multipliers with fixed input field size and varying output field size. As anticipated, the number of nodes has increased owing to the possible lack of sharing.

Table 11: O/P field size varying with constant Input field size of 2.

Multiplier	SBDD	GF(2^2)	GF(2^3)	GF(2^4)
2*2	14	15	11	12
3*3	51	50	65	46
4*4	157	178	194	260
5*5	471	490	553	755
6*6	1348	1587	1917	1890

As we move to a higher order field from a lower order field, the number of nodes usually decrease. This decrease is also associated with fewer number of levels and shorter path lengths compared to conventional BDDs and their variants.

Evaluation Time Table 12 shows the results for the APLs as compared to SBDDs. For the majority of the cases the APLs are significantly lower than those in the SBDDs. Also, as we go from GF(4) to GF(8), the APLs reduce further. On an average the APLs are 3 times less in GF(4), while about 6 times less in GF(8) as compared to the SBDDs. I.e. the evaluation time is essentially halving as we go from GF(4) to GF(8). Note that the number of nodes in the SMODDs is almost identical on an average compared to those in the SBDDs. This is owing to the fact that the SMODDs have been ordered based on the APLs, which does not necessarily guarantee reduced node count. The dashes ('-') in the table indicate that results for those circuits (i.e. ex1010, ex5, b12, risc, apex4) are not available for the BDDs.

Tables 13 and 14 present comparison of the APLs for SMODD, CF-SMODD, and ECF-SMODD for the benchmark circuits modeled in GF(4) and GF(8) respectively. These tables also show the results for 50,000 random vectors. The results for random pattern simulation constitutes the net total path lengths for the 50,000 vectors. The spatial complexity is reflected by the node count and the speed of evaluation by the APLs and random pattern simulations. These results reflect speedup over current methods for simulation such as in [95]. The trade off between these two factors across the representations is clearly evident from the results shown in these tables. In general it can be seen that the CF-SMODD clearly wins out in terms of speed whereas the ECF-SMODD tries to optimize between the speed and node count.

5.12 Conclusions

This chapter focused on a framework for representing multiple-output binary and word-level circuits based on canonic DDs in GF(N). We showed that such reduced ordered DDs are canonical and minimal with respect to a fixed variable ordering. Techniques for further node and path optimization have also been presented. We also presented the theory for representing functions in GF(N) in terms of their CF and ECF under the same framework.

The proposed DDs have been tested on many benchmarks with varying input and output field sizes. The results suggest superior performance in terms of node compression as well as reduced APLs, which implies improved evaluation times. This has also been confirmed by simulation of 50,000 randomly selected vectors. Overall the results seem to suggest that the proposed framework can serve as an effective medium for verification as well as for simulation, testing, and safety checking.

Table 12: APL compared to SBDD.

Benchmark	I/P	O/P	BDD		SMODD					
			Nodes	APL	GF(4)			GF(8)		
					Nodes	APL	% Imp.	Nodes	APL	% Imp.
duke2	22	29	366	150.3	793	80.61	186	783	48.87	307
cordic	23	2	-	-	29	4.41	-	21	3.29	-
misex2	25	18	100	75.6	93	16.55	456	50	8.61	877
vg2	25	8	90	48.9	892	24.15	202	867	17.2	284
table5	17	15	685	114.1	751	32.12	355	635	16.02	712
pdc	16	40	596	215.4	433	52.58	409	384	33.13	650
e64	65	65	194	256	943	64.96	394	858	56.27	454
spla	16	46	628	226.6	352	47.33	478	261	30.18	750
ex1010	10	10	-	-	662	24.69	-	346	14.74	-
ex5	8	63	-	-	194	66.72	-	144	36.05	-
b12	15	9	-	-	78	15.34	-	48	8.31	-
x6dn	39	5	235	41.2	212	7.26	567	165	4.09	100
exep	30	63	675	255.7	462	53.86	474	409	33.44	764
x1dn	27	6	139	41	151	10.3	397	120	6.56	624
x9dn	27	7	139	50.4	160	12.6	399	174	10.66	472
mark1	20	31	119	115.7	149	36.95	313	150	26.63	434
mainpla	27	54	1857	277.5	2414	138.26	200	1667	70.74	392
risc	8	31	-	-	58	23.72	-	32	13.33	-
xparc	41	73	1947	304.8	1925	94.46	322	1550	58.22	523
apex4	9	19	-	-	536	36.54	-	324	16.73	-
Average	23.5	29.7	555	155.23	564.35	42.17	368	449.45	25.65	524.5

Acknowledgments

The work on TEDs described in this chapter has been supported by a grant from the National Science Foundation under award No. CCR-0204146, INT-0233206 and CCR-0702506. The work on FFDDs have been funded, in part, by the Engineering and Physical Science Research Council, UK Grant NO. GR/S40855/01

References

[1] R.E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Trans. Comput.*, vol. C–35, no. 8, pp. 677–691, Aug. 1986.

[2] R.E. Bryant and Y.A. Chen, "Verification of Arithmetic Functions with Binary Moment Diagrams," in *DAC-95*, 1995.

[3] M. Ciesielski, P. Kalla, and S. Askar, "Taylor Expansion Diagrams: A Canonical Representation for Verification of Data Flow Designs," *IEEE Trans. on Computers*, vol. 55, no. 9, pp. 1188–1201, Sept. 2006.

[4] A.M. Jabir, D.K. Pradhan, T.L. Rajaprabhu, and A.K. Singh, "A Technique for Representing Multiple Output Binary Functions with Applications to Verification and Simulation," *IEEE Trans. on Computers*, 2007.

Table 13: APL with random pattern simulation in GF(4).

Benchmark	SMODD			CF-SMODD			ECF-SMODD		
(IP/OP)	Nodes	APL	Random Simulation	Node	APL	Random Simulation	Nodes	APL	Random Simulation
duke2 (22/29)	793	80.61	4023605	445	8.1	405126.5	470	28.03	439206.5
cordic (23/2)	29	4.41	220771.5	30	5.14	256775	28	4.17	83466.8
misex2 (25/18)	93	16.55	825155	131	6.65	317410	83	4.22	84350.6
table5 (17/15)	751	32.12	1605105	555	6.51	326386.5	604	14.67	201920
pdc (16/40)	433	52.58	2630225	1356	9.28	464182	418	63.54	1200000
e64 (65/65)	943	64.96	3251360	66	1.33	66643.5	442	13.24	226700
spla (16/46)	352	47.33	2384420	1454	7.26	356000	345	47.93	961395
ex1010 (10/10)	662	24.69	1234420	555	7.29	343225	655	36.55	600000
ex5 (8/63)	192	66.72	3337715	980	20.77	1083580	205	110.72	2200000
b12 (15/9)	78	15.34	767230	157	7.11	265646.5	62	21.6	400000
x6dn (39/5)	212	7.26	363796.5	201	5.4	265861.5	144	3.08	5012.8
exep (30/63)	462	53.86	2695070	1459	10.05	493531.5	423	23.75	509220
x1dn (27/6)	151	10.3	515495	358	6.73	332395	127	5.92	2.71
x9dn (27/7)	160	12.6	628695	456	7.95	397483	129	6.47	129333.4
mark1 (20/31)	149	36.95	1850300	281	8.69	426828.5	116	18.4	323713.5
mainpla (27/54)	2414	138.26	6914400	2153	9.56	468033.5	811	41.51	1083405
risc (8/31)	58	23.72	1185410	64	4.97	248535	55	14.91	215953.5
xparc (41/73)	1925	94.45	4732915	2537	5.31	260840	983	16.63	337326.5
apex4 (9/19)	536	36.54	1825970	1078	10.81	540455	505	17.7	204093.5

[5] G. DeMicheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, 94.

[6] R. Anderson, “An Introduction to Binary Decision Diagrams,” http://www.itu.dk/people/hra/notes-index.html, 1997.

[7] G. Hachtel and F. Somenzi, *Logic Synthesis and Verification Algorithms*, Kluwer Academic Publishers, 1998.

[8] R. Rudell, “Dynamic Variable Ordering for Ordered Binary Decision Diagrams,” in *IEEE International Conference on Computer-Aided Design*, 1993, pp. 42–47.

[9] O. Coudert and J.C. Madre, “A Unified Framework for the Formal Verification of Sequential Circuits,” in *Proc. ICCAD*, 1990, pp. 126–129.

[10] H.J. Touati, H. Savoj, B. Lin, R.K. Brayton, and A. Sangiovanni-Vincentelli, “Implicit State Enumeration of Finite State Machines using BDDs,” in *Proc. ICCAD*, 1990, pp. 130–133.

[11] E. A. Emerson, “Temporal and Modal Logic,” in *Formal Models and Semantics*, J. van Leeuwen, Ed., vol. B of *Handbook of Theoretical Computer Science*, pp. 996–1072. Elsevier Science, 1990.

[12] K. L. McMillan, *Symbolic Model Checking*, Kluwer Academic Publishers, 1993.

[13] R. E. Bryant and Y-A. Chen, “Verification of Arithmetic Functions with Binary Moment Diagrams,” in *Proc. Design Automation Conference*, 1995, pp. 535–541.

Table 14: APL with random pattern simulation in GF(8).

Benchmark (IP/OP)	SMODD			CF-SMODD			ECF-SMODD		
	Nodes	APL	Random Simulation	Node	APL	Random Simulation	Nodes	APL	Random Simulation
duke2 (22/29)	783	48.87	2444800	373	6.53	326456.5	412	29.15	637575
cordic (23/2)	21	3.29	165215	22	3.68	184286.5	20	2.7	90145
misex2 (25/18)	50	8.61	429593.5	88	4.2	204385	43	16.63	400000
table5 (17/15)	635	16.02	801950	379	4.48	224104.5	247	4.28	207866.5
pdc (16/40)	384	33.13	1656360	1096	5.62	281190	387	57.13	1200000
e64 (65/65)	858	56.27	2813180	44	1.14	57121.5	443	3.01	75326.25
spla (16/46)	261	30.18	1509950	1132	3.77	188649	259	31.6	848145
ex1010 (10/10)	346	14.74	737095	337	5.8	274060	345	22.73	400000
ex5 (8/63)	144	36.05	1802605	652	13.28	663865	148	68.05	1600000
b12 (15/9)	48	8.31	415075	120	5.43	265646.5	46	16.05	400000
x6dn (39/5)	165	4.09	206140	232	2.71	134725	151	2.45	81764.33
exep (30/63)	409	33.44	1674265	1073	6.54	327246.5	368	157.18	4536146.5
x1dn (27/6)	120	6.56	327983.5	302	5.23	259120	121	14.56	400000
x9dn (27/7)	174	10.66	532785	216	5.27	263295.5	129	5.16	129103.75
mark1 (20/31)	150	26.63	1850300	196	6.53	332668.5	116	139.87	800400
mainpla (27/54)	1667	70.74	3545165	1593	6.69	329915	537	91.71	2764640
risc (8/31)	32	13.33	666330	44	3.1	154980.5	35	27.16	800000
xparc (41/73)	1550	58.22	2914670	1820	3.8	188025	731	3.22	59733.25
apex4 (9/19)	324	16.73	837530	722	7.46	373047	325	24.73	400000

[14] K. S. Brace, R. Rudell, and R. E. Bryant, “Efficient Implementation of the BDD Package,” in *DAC*, 1990, pp. 40–45.

[15] F. Somenzi, *CUDD: CU Decision Diagram Package*, Univ. of Colorado at Boulder, USA, `http://vlsi.colorado.edu/~fabio/CUDD/`, Release 2.3.0, 1998.

[16] R. K. Brayton, G. D. Hachtel, A. Sangiovanni-Vencentelli, F. Somenzi, A. Aziz, S-T. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. Ranjan, S. Sarwary, G. Shiple, S. Swamy, and T. Villa, “Vis: A system for verification and synthesis,” *Proceedings of the Computer Aided Verification Conference*, pp. 428–432, 1996.

[17] U. Kebschull, E. Schubert, and W. Rosentiel, “Multilevel Logic Synthesis based on Functional Decision Diagrams,” in *EDAC*, 1992, pp. 43–47.

[18] R. Drechsler, A. Sarabi, M. Theobald, B. Becker, and M.A. Perkowski, “Efficient Representation and Manipulation of Switching Functions based on Order Kronecker Function Decision Diagrams,” in *Proc. Design Automation Conference*, 1994, pp. 415–419.

[19] S. Minato, “Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems,” in *Proc. Design Automation Conference*, 1993, pp. 272–277.

[20] R. E. Bryant, “Binary Decision Diagrams and Beyond: Enabling Technologies for Formal Verificaiton,” in *Intl. Conferene on Computer-Aided Design*, 1995.

[21] S. Horeth and Drechsler, “Formal Verification of Word-Level Specifications,” in *DATE*, 1999, pp. 52–58.

[22] E. M. Clarke, K. L. McMillan, X. Zhao, M. Fujita, and J. Yang, "Spectral Transforms for Large Boolean Functions with Applications to Technology Mapping," in *DAC*, 93, pp. 54–60.

[23] I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi, "Algebraic Decision Diagrams and their Applications," in *ICCAD*, Nov. 93, pp. 188–191.

[24] Y-T. Lai and S. Sastry, "Edge-valued Binary Decision Diagrams for Multi-level Hierarchical Verification," in *Proc. Design Automation Conference*, 1992, pp. 608–613.

[25] Y-T. Lai, M. Pedram, and S. B. Vrudhula, "FGILP: An ILP Solver based on Function Graphs," in *ICCAD*, 1993, pp. 685–689.

[26] Y-A. Chen and R. Bryant, "PHDD: An Efficient Graph Representation for Floating Point Circuit Verification," in *IEEE Int. Conference on Computer-Aided Design*, 1997, pp. 2–7.

[27] R. Drechsler, B. Becker, and S. Ruppertz, "The K*BMD: A Verification Data Structure," *IEEE Design & Test*, pp. 51–59, 1997.

[28] H. Enderton, *A mathematical Introduction to Logic*, Academic Press New York, 1972.

[29] T. Bultan and et. al, "Verifying Systems with Integer Constraints and Boolean Predicates: a Composite Approach," in *Proc. Int'l. Symp. on Software Testing and Analysis*, 1998.

[30] S. Devadas, K. Keutzer, and A. Krishnakumar, "Design Verification and Reachability Analysis using Algebraic Manipulation," in *Proc. Intl. Conference on Computer Design*, 1991.

[31] G. Ritter, "Formal Verification of Designs with Complex Control by Symbolic Simulation," in *Advanced Research Working Conf. on Correct Hardware Design and Verification Methods (CHARME)*, Springer Verlag LCNS, Ed., 1999.

[32] R. E. Shostak, "Deciding Combinations of Theories," *Journal of ACM*, vol. 31, no. 1, pp. 1–12, 1984.

[33] Aaron Stump, Clark W. Barrett, and David L. Dill, "CVC: A Cooperating Validity Checker," in *14th International Conference on Computer Aided Verification (CAV)*, Ed Brinksma and Kim Guldstrand Larsen, Eds. 2002, vol. 2404 of *Lecture Notes in Computer Science*, pp. 500–504, Springer-Verlag, Copenhagen, Denmark.

[34] M. Chandrashekhar, J. P. Privitera, and J. W. Condradt, "Application of term rewriting techniques to hardware design verification," in *Proc. Design Automation Conf.*, 1987, pp. 277–282.

[35] Z. Zhou and W. Burleson, "Equivalence Checking of Datapaths Based on Canonical Arithmetic Expressions," in *Proc. Design Automation Conference*, 1995.

[36] S. Vasudevan, "Automatic Verification of Arithmetic Circuits in RTL using Term Rewriting Systems," M.S. thesis, University of Texas, Austin, 2003.

[37] J. Burch and D. Dill, "Automatic Verification of Pipelined Microprocessor Control," in *Computer Aided Verification*. LCNS, July 1994, Springer-Verlag.

[38] R. Bryant, S. German, and M. Velev, “Processor Verification using Efficient Reductions of the Logic of Uninterpreted Functions to Propositional Logic,” *ACM Trans. Computational Logic*, vol. 2, no. 1, pp. 1–41, 2001.

[39] M. Velev and R. Bryant, “Effective use of Boolean Satisfiability Procedures in the Formal Verification of Superscalar and VLIW Microprocessors,” *Journal of Symbolic Computation*, vol. 35, no. 2, pp. 73–106, 2003.

[40] R. Bryant, S. Lahiri, and S. Seshia, “Modeling and Verifying Systems using a Logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions,” *CAV*, 2002.

[41] A. Goel and *et al.*, “BDD Based Procedures for a Theory of Equality with Uninterpreted Functions,” *CAV*, pp. 244–255, 1998.

[42] L. Arditi, “*BMDs can Delay the use of Theorem Proving for Verifying Arithmetic Assembly Instructions,” in *In Proc. Formal Methods in CAD (FMCAD)*, Srivas, Ed. 1996, Springer-Verlag.

[43] M. Moskewicz, C. Madigan, L. Zhang Y. Zhao, and S. Malik, “Chaff: Engineering an efficient SAT solver,” in *Proc. of 38th Design Automation Conf.*, June 2001, pp. 530–535.

[44] E. Goldberg and Y. Novikov, “BerkMin: A Fast and Robust Sat-Solver,” in *Proc. Design Automation and Test in Europe, DATE-02*, 2002, pp. 142–149.

[45] C.-Y. Huang and K.-T. Cheng, “Using Word-Level ATPG and Modular Arithmetic Constraint Solving Techniques for Assertion Property Checking,” *IEEE Trans. CAD*, vol. 20, pp. 381–391, 2001.

[46] M. Iyer, “RACE: A word-level ATPG-based Constraints Solver System for Smart Random Simulation,” in *Internationall Test Conf., ITC-03*, 2003, pp. 299–308.

[47] R. Brinkmann and R. Drechsler, “RTL-Datapath Verification using Integer Linear Programming,” in *Proc. ASP-DAC*, 2002.

[48] Z. Zeng, P. Kalla, and M. Ciesielski, “LPSAT: A unified approach to RTL satisfiability,” in *Proc. DATE*, March 2001, pp. 398–402.

[49] F. Fallah, S. Devadas, and K. Keutzer, “Functional Vector Generation for HDL Models using Linear Programming and 3-Satisfiability,” in *Proc. Design Automation Conference*, 1998, pp. 528–533.

[50] G. Bioul and M. Davio, “Taylor Expansion of Boolean Functions and of their Derivatives,” *Philips Research Reports*, vol. 27, no. 1, pp. 1–6, 1972.

[51] A. Thayse and M. Davio, “Boolean Differential Calculus and its Application to Switching Theory,” *IEEE Trans. on Comp.*, vol. C-22, no. 4, pp. 409–420, 1973.

[52] Maple, ,” http://www.maplesoft.com.

[53] Mathematica, ,” http://www.wri.com.

[54] The MathWorks, "Matlab," http://www.mathworks.com.

[55] M. Ganai, L. Zhang, P. Ashar, A. Gupta, and S. Malik, "Combining strengths of circuit-based and CNF-based algorithms for a high-performance SAT solver," in *Design Automation Conference (DAC-2002)*, June 2002, pp. 747–750.

[56] Z. Zeng, K. Talupuru, and M. Ciesielski, "Functional Test Generation based on Word-level SAT," in *Journal of Systems Architecture*. Aug. 2005, vol. 5, pp. 488–511, Elsevier Publishers.

[57] R. E. Bryant and Y-A. Chen, "Verification of Arithmetic Functions with Binary Moment Diagrams," in *Design Automation Conference*, 1995, pp. 535–541.

[58] Y. A. Chen and R. E. Bryant, "*PHDD: An Efficient Graph Representation for Floating Point Verification," in *Proc. ICCAD*, 1997.

[59] D. Stoffel and W. Kunz, "Equivalence Checking of Arithmetic Circuits on the Arithmetic Bit Level," *IEEE Trans. on CAD*, vol. 23, no. 5, pp. 586–597, May 2004.

[60] N. Shekhar, P. Kalla, F. Enescu, and S. Gopalakrishnan, "Equivalence Verification of Polynomial Datapaths with Fixed-Size Bit-Vectors using Finite Ring Algebra," in *Intl. Conf. on Computer-Aided Design, ICCAD*, 2005.

[61] P. Sanchez and S. Dey, "Simulation-Based System-Level Verification using Polynomials," in *High-Level Design Validation & Test Workshop, HLDVT*, 1999.

[62] R. Drechsler, *Formal Verification of Circuits*, Kluwer Academic Publishers, 2000.

[63] Y. Lu, A. Koelbl, and A. Mathur, "Formal Equivalence Checking between System-level Models and RTL, embedded tutorial," in *Intl. Conference on CAD (ICCAD'05)*, 2005.

[64] P. Georgelin and V. Krishnaswamy, "Towards a C++-based Design Methodology Facilitating Sequential Equivalence Checking," in *Design Automation Conf. (DAC'06*, 2006, pp. 93–96.

[65] D. Brier and R.S. Mitra, "Use of C/C++ Models for Architecture Exploration and Verification of DSPs," in *Design Automation Conf. (DAC'06)*, 2006, pp. 79–84.

[66] Calypto Design Systems, "Verification without Testbenches," http://www.calypto.com.

[67] A. Vellelunga and D. Giramma, "The Formality Equivalence Checker provides industry's best Arithmetic Verification Coverage," *Verification Avenue, Synopsys Technical Buletin*, vol. 5, no. 2, pp. 5–9, June 2004.

[68] E. Kryrszig, *Advanced Engineering Mathematics*, John Wiley and Sons, Inc., 1999.

[69] F. Winkler, *Polynomial Algorithms in Computer Algebra*, Springer, 1996.

[70] M. Ciesielski, P. Kalla, Z. Zeng, and B. Rouzeyre, "Taylor Expansion Diagrams: A Compact Canonical Representation with Applications to Symbolic Verification," in *Design Automation and Test in Europe*, 2002, pp. 285–289.

[71] Université de Bretagne Sud LESTER, "GAUT, Architectural Synthesis Tool," http://lester.univ-ubs.fr:8080, 2004.

[72] F. Somenzi, "Colorado Decision Diagram Package," Computer Programme, 1997.

[73] R. Rudell, "Dynamic Variable Ordering for Binary Decision Diagrams," in *Proc. Intl. Conf. on Computer-Aided Design*, Nov. 1993, pp. 42–47.

[74] D. Gomez-Prado, Q. Ren, S. Askar, M. Ciesielski, and E. Boutillon, "Variable Ordering for Taylor Expansion Diagrams," in *IEEE Intl. High Level Design Validation and Test Workshop, HLDVT-04*, 2004, pp. 55–59.

[75] M. Ciesielski, S Askar, D. Gomez-Prado, J. Guillot, and E. Boutillon, "Data-Flow Transformations using Taylor Expansion Diagrams," in *Design Automation and Test in Europe*, 2007, pp. 455–460.

[76] P. Jain, "Parameterized Motion Estimation Architecture For Dynamically Varying Power and Compression Requirements," M.S. thesis, Dept. of Electrical and Computer Engineering, University of Massachusetts, 2002.

[77] D. Pradhan, S. Askar, and M. Ciesielski, "Mathematical Framework for Representing Discrete Functions as Word-level Polynomials," in *IEEE Intl. High Level Design Validation and Test Workshop, HLDVT-03*, 2003, pp. 135–139.

[78] W. Stallings, *Cryptography and Network Security*, Prentice Hall, Englwood Cliffs, N.J., 1999.

[79] S.B. Wicker, *Error Control Systems for Digital Communication and Storage*, Prentice Hall, Englwood Cliffs, N.J., 1995.

[80] R.E. Blahut, *Fast Algorithms for Digital Signal Processing*, Addison-Wesley, Reading, Mass., 1984.

[81] T. Kam, T. Villa, R.K. Brayton, and A.L. Sangiovanni-Vincentelli, "Multi-valued Decision Diagrams: Theory and Applications," *Multiple Valued Logic*, vol. 4, no. 1–2, pp. 9–62, 1998.

[82] D.M. Miller and R. Drechsler, "On the Construction of Multiple-Valued Decision Diagrams," in *Proc. 32nd ISMVL*, Boston, USA, 2002, pp. 245–253.

[83] C. Scholl, R. Drechsler, and B. Becker, "Functional Simulation using Binary Decision Diagrams," in *Int. Conf. Comp. Aided. Des. (ICCAD'97)*, 1997, pp. 8–12.

[84] Y. Jiang and R.K. Brayton, "Software Synthesis from Synchronous Specification using Logic Simulation Techniques," in *Des. Automat. Conf. (DAC'02)*, New Orleans, LA, USA, June 2002, pp. 319–324.

[85] D.K. Pradhan, "A Theory of Galois Switching Functions.," *IEEE Trans. Comp.*, vol. C–27, no. 3, pp. 239–249, Mar. 1978.

[86] K.M. Dill, K. Ganguly, R.J Safranek, and M.A. Perkowski, “A New Zhegalkin Galois Logic,” in *Reed-Müller–97*, Oxford, UK, Sept. 1997, pp. 247–257.

[87] C.H. Wu, C.M. Wu, M.D. Sheih, and Y.T. Hwang, “High-Speed, Low-Complexity Systolic Design of Novel Iterative Division Algorithm in GF(2^m),” *IEEE Trans. Comput.*, vol. 53, pp. 375–380, Mar. 2004.

[88] P.C. McGeer, K.L. McMillan, and A.L. Sangiovanni-Vincentell, “Fast Discrete Function Evaluation using Decision Diagram,” in *Int. Conf. Comp. Aided. Des. (ICCAD’95)*, Nov. 1995, pp. 402–407.

[89] M.J. Ciesielski, P. Kalla, Z. Zeng, and B. Rouzeyere, “Taylor Expansion Diagrams: A Compact, Canonical Representation with Applications to Symbolic Verification,” in *Design Automation and Test in Europe*, Mar. 2002.

[90] A. Jabir and D. Pradhan, “MODD: A New Decision Diagram and Representation for Multiple Output Binary Functions,” in *Des. Automat. Test. in Europe (DATE’04)*, Paris, France, Feb. 2004, pp. 1388–1389.

[91] R.S. Stanković and R. Dreschler, “Circuit Design from Kronecker Galois Field Decision Diagrams for Multiple-Valued Functions,” in *ISMVL-27*, May 1997, pp. 275–280.

[92] D.K. Pradhan, M. Ciesielski, and S. Askar, “Mathematical Framework for Representing Discrete Functions as Word-Level Polynomials,” in *Proc. HLDVT’03, San Fransisco, USA.*, Nov. 2003, pp. 135–142.

[93] P. Asher and S. Malik, “Fast Functional Simulation Using Branching Programmes,” in *ICCAD’95*, Oct. 1995, pp. 408–412.

[94] Jon T. Butler, T. Sasao, and M. Matsuura, “Average Path Length of Binary Decision Diagrams,” *IEEE Trans. Comput.*, vol. 54, no. 9, pp. 1041–1053, Sept. 2005.

[95] T. Sasao, Y. Iguchi, and M. Matsuura, “Comparison of Decision Diagrams for Multiple-Output Logic Functions,” in *Proc. IWLS*, 2002.

[96] A. Reyhani-Masoleh and M.A. Hasan, “Low Complexity Bit Parallel Architectures for Polynomial Basis Multiplication over GF(2^m),” *IEEE Trans. Comp.*, vol. 53, no. 8, pp. 945–959, Aug. 2004.

[97] S. Minato, “Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems,” in *Proc. 30th IEEE/ACM Des. Automation Conf. (DAC’93)*, June 1993, pp. 272–277.

[98] A. Jabir, T. Rajaprabhu, D. Pradhan, and A. Singh, “MODD For CF: A Compact Representation for Multiple-Output Functions,” in *Proc. Int. Conf. High Level Des. Val. Test (HLDVT’04)*, California, USA, Nov. 2004.

[99] F. Schmiedle, W. Gunther, and R. Drechsler, “Selection of Efficient Re-Ordering Heuristics for MDD Construction,” in *Proc. 31st IEEE Int. Symp. on Multi-Valued Logic (ISMVL’01)*, 2001, pp. 299–304.