Boolean Function Representation Using Parallel-Access Diagrams.*

Valeria Bertacco and Maurizio Damiani
Dipartimento di Elettronica ed Informatica
Universitá di Padova, Via Gradenigo 6/A, 35131 Padova, ITALY

Abstract

In this paper we introduce a nondeterministic counterpart to Reduced, Ordered Binary Decision Diagrams for the representation and manipulation of logic functions. ROBDDs are conceptually related to deterministic finite automata (DFA), accepting the language formed by the minterms of a function. This analogy suggests the use of nondeterministic devices as language recognizers. Unlike ROBDDs, the diagrams introduced in this paper allow multiple outgoing edges with the same label. By suitably restricting the degree of nondeterminism, we still obtain a canonical form for logic functions.

Using PADs, we are able to reduce the memory occupation with respect to traditional ROBDDs for several benchmark functions. Moreover, the analysis of the PAD graphs allowed us to sometimes identify new and better variable ordering for several benchmark circuits.

1 Introduction

Reduced, Ordered Binary Decision Diagrams (ROBDDs) [1, 2] are the most CPU- and memory- efficient data structure known so far for the manipulation of large logic functions. For this reason, they are becoming pervasive in logic synthesis and verification environments [3, 4, 5, 6]. Ongoing research is attempting to extend their applicability to other domains, such as the solution of graph problems and integer-linear programming [7, 8]. ROBDDs, however, are not exempt from inefficiencies. Classes of functions exist for which they may require an exponential amount of memory. This may be due either to the intrinsic nature of the functions (*i.e.* multiplication, hidden weighted bit, clique-related functions, etc ... [9, 10]), or to an improper ordering of variables. These inefficiencies motivate a substantial amount of research on the development of ordering heuristics [11, 12] and alternative representations [13, 14].

ROBDDs are conceptually related to Deterministic Finite Automata. In particular, the ROBDD of a Boolean function F can be obtained from the minimum-state DFA accepting the language formed by the set of minterms of F, simply by removing the so-called **redundant-test** states. This relationship is further described in Section (2).

In this paper we demonstrate the possibility of using data structures related with Nondeterministic Finite Automata (NFA) as alternative representations. The paper introduces Parallel-Access Diagrams, a nondeterministic analog to ROBDDs. In a PAD, a vertex can have multiple outgoing edges **with the same label**. Thus, in a PAD, from one such vertex, we can access in parallel multiple vertices. Following the semantics of NFA, the function that is globally pointed to by the set of pointers is the logic OR of the functions pointed by each pointer. We show that by suitably restricting the choices of multiple pointers, a function can be represented by a unique PAD. We can thus retain the canonicity properties of ROBDDs. We then show how to manipulate Boolean functions using PADs, and provide experimental results on several benchmark circuits.

On the theoretical front, we show that PADs are more compact and more robust than ROBDDs: classes of functions exist with polynomially-sized PADs, but only exponentially-sized ROBDDs, and some order-sensitive functions (with worst-case exponential ROBDD size) may have an order-insensitive, linear-size PAD representation. This property can be used also for deriving better variable orderings for BDDs. Moreover, in no case the size of a ROBDD can be smaller than that of a PAD.

For reasons of space, in this paper we do not include proofs of theorems, but we will make them available upon request.

2 Function representations and automata.

In this section, we briefly review previous work in the area of function representations. We focus on the relationship of automata with ROBDDs and covers. We assume some familiarity with basic notions of automata theory.

2.1 ROBDDs and DFA

In [1], ROBDDs are introduced as a "processor", receiving 1 bit of data at a time and issuing the value of F as output after a sufficient number of data bit are entered. The "processor" is conceptually very close to a DFA, accepting a finite set of strings - the minterms of F -, and issuing a logic "1" upon acceptance.

Example 1. Fig. (1.a) shows the truth table of a 4-variable function F. The input combinations resulting in a 1 of F form a set of strings, each string of length 4. This set, in turn, describes completely F. Fig. (1.a) shows the state-minimal automaton accepting this set of strings. Since all strings have the same length, the automaton can be factored in two parts. The first part is a counter, sequencing out variable symbols. The second automaton takes as inputs $(variable, value)$ pairs. It is easily verifiable that this second automaton corresponds precisely to the BDD of F. The factorization and the final BDD are reported in Fig. (1.b) and (1.c), respectively. □

[0]This research was partially supported by the ESPRIT III Basic Research Programme of the EC under contract No. 9072 (Project GEPPCOM) and by CNR grant # 95.02061.CT07

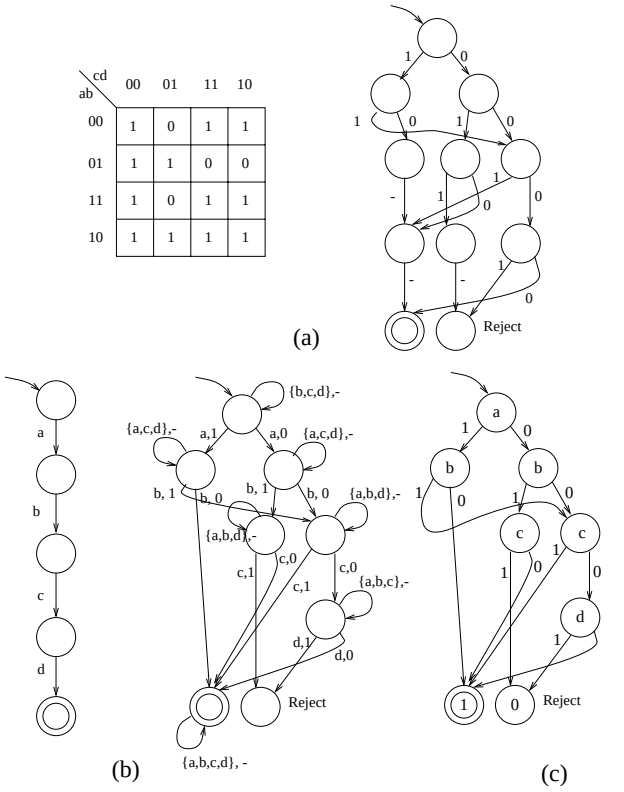

Figure 1. a) Truth table of a 4-variable function and the DFA accepting the minterms of its ON set. b) Decomposition of the DFA into a counter and a second machine. c) the ROBDD of the function.

2.2 Parallel-Access Diagrams and NFA.

Given an arbitrary cover of a function F, it is possible to construct a **nondeterministic** automaton accepting the minterms of F. Redundant test states can be removed, just like for ROBDDs, to obtain another directed acyclic graph for the function. Example (2) below exemplifies this construction.

Example 2. Consider the the same function as in Example (1): $F = a'bc' + ab' + ac + b'c + c'd'$. A nondeterministic automaton accepting precisely the minterms of F is shown in Fig. (2.a). The set of start states is identified by the set of start (dashed) edges. Removal of redundant test states, identification of isomorphic subgraphs, and labeling of nodes produces the graph of Fig. (2.b). □

The resulting graph may present some key differences from a ROBDD: Vertices may have multiple outgoing edges with the same label, and the graph itself may have multiple roots.

Definition 1. *A PAD is a multi-rooted, directed acyclic graph, with two distinct sink vertices, labeled 0 and 1, respectively. All other vertices* **v** *are labeled by a Boolean variable, and have two non-empty sets of outgoing edges, denoted by* **0***(v) and* **1***(v). Moreover, each path in the graph must be* **consistent** *with a pre-defined variable ordering. A PAD is termed* **reduced** *if it contains no two isomorphic subgraphs.* □

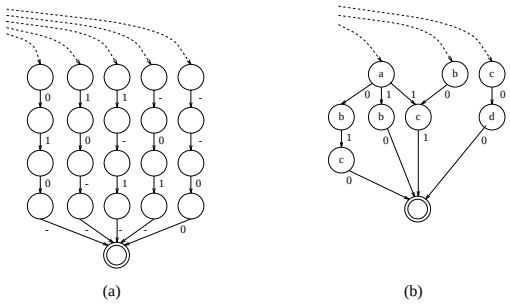

Figure 2. a) NFA accepting the minterms of the function of Example (2). The NFA of part (a) after BDD-like simplifications. Reject conditions are removed, for the sake of simplicity.

A PAD defines recursively a logic function by the following rules:

- Vertices 0 and 1 denote the constant functions 0 and 1, respectively.
- A vertex v, with Boolean variable label x, defines the logic function $f_v = \overline{x} F_{\mathbf{0}(v)} + x F_{\mathbf{1}(v)}$
- A set $S = \{v_1, \cdots, v_n\}$ of vertices defines a logic function

$$F_S = \bigcup_{v_i \in S} f_{v_i} \quad (1)$$

The following example points out the potential advantages of PADs over ROBDDs.

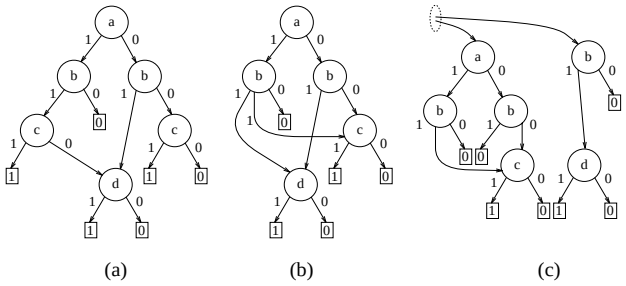

Figure 3. a) The ROBDD of a 4-variable function; b) a PAD representation; c) another PAD.

Example 3. Fig. (3.a) shows the ROBDD of the function $F = a'(b'c + bd) + ab(c + d)$, with a lexicographical ordering of the variables. Since the four residue functions $F_{ab}, F_{a'b}, F_{ab'}, F_{a'b'}$ are all distinct, the uppermost two levels of the graph form a binary tree. Notice, however, that $F_{ab} = c + d = F_{a'b} + F_{a'b'}$. We can express this property of F by replacing each edge pointing to F_{ab} by a pair of edges, pointing to $F_{a'b}$ and $F_{a'b'}$, respectively. This transformation is shown in Fig. (3.b). □

By expressing F_{ab} as the sum of other already existing functions, we are able to save the construction of the

ROBDD for F_{ab}. In the small example of Fig. (3), this saves us one vertex, at the expense of adding one extra pointer. Unfortunately, unlike ROBDDs, there may be several PAD representations of a logic function F, using the same variable ordering: as an example, Fig. 3 shows three different constructions of the same function using PADs.

In the next section we show that, by reducing the type of nondeterminism allowed at each vertex, it is possible to obtain a canonical representation.

3 Disjoint-support OR decompositions.

If a function f is represented by a multi-rooted PAD, the functions pointed by each root form a cover of f. The multiplicity of PADs for representing f reflect the multiplicity of covers for f. In this paper, we constrain the type of covers that can be generated, in such a way that we can grant the uniqueness of the generated PAD. To this end we leverage upon the following simple but useful definitions and results on logic decomposition [15, 16].

Definition 2. *Let $f : B^n \rightarrow B$ denote a non-constant Boolean function of n variables $x_i; i = 1, \cdots, n$. We say that f* **depends** *on x_i if $\partial f / \partial x_i$ is not identically 0. We call* **support** *of f (indicated by $S(f)$) the set of Boolean variables f depends on.* □

Definition 3. *A set of non-constant functions $\{f_1, \cdots f_k\}$, with respective supports $S(f_i)$ is called a* **disjoint-support OR decomposition** *of f if*

$$\sum_{i=1}^{k} f_i = f; \qquad S(f_i) \cap S(f_j) = \emptyset, \;\; i \neq j \tag{2}$$

A disjoint support OR decomposition is **maximal** *if no function f_i has further decomposition. We indicate by $DSOD(f)$ such a maximal decomposition.* □

Theorem 1. *There is a unique maximal disjoint-support OR decomposition of a function.* □

Definition 4. *Consider a vertex v of a PAD, and let $e_1, \cdots, e_n$ denote n outgoing edges from v with the same label. Let also f_i denote the function pointed by each edge e_i. We say that a PAD has the* **disjoint-support property** *if for each vertex v, the set of functions $\{f_1, \cdots, f_n\}$ is a DSOD.* □

Example 4. Consider the three PADs of Fig. (3). Only the second one satisfies the disjoint-support property. In the third PAD, the two functions pointed by the root edges share the variable b. □

Theorem 2. *For a given function f and variable ordering, there is a unique PAD representation with the DSOD property.* □

The proof follows intuitively from the uniqueness of a DSOD decomposition.

3.1 Properties of DSODs.

In the rest of the paper, we focus on PADs with the **disjoint-support** property. We conclude the section by pointing some results on DSODs that are useful for the construction of PAD manipulation routines.

Theorem 3. *Suppose $\{f_1, \cdots, f_k\}$ is a DSOD of some function. Then, by erasing elements from the set, the new set is also a DSOD.* □

Theorem 4. *If $DSOD(f) = \{f_1, \cdots, f_k\} \cup \{p_1, \cdots, p_h\}$ and $DSOD(g) = \{g_1, \cdots, g_l\} \cup \{p_1, \cdots, p_h\}$, where $g_i \neq f_j, i = 1, \cdots l, j = 1, \cdots k$, then:*

1. $DSOD(f \cdot g) =$
   $\{p_1, \cdots, p_h\} \cup DSOD((f_1 + ... + f_k) \cdot (g_1 + ... + g_l))$.
2. $DSOD(f + g) =$
   $\{p_1, \cdots, p_h\} \cup DSOD(f_1 + ... + f_k + g_1 + ... + g_l)$
3. *Let x denote a variable not in the support of f or g. Then:*
   $DSOD(x'f + xg) =$
   $\{p_1, \cdots, p_h\} \cup \{x'(f_1 + ... + f_k) + x(g_1 + \cdots + g_l)\}$

□

Theorem 5. *Let x denote a variable, $x \notin S(g)$, and suppose $f = x + g$. Then,*

$$DSOD(f) = \{x\} \cup DSOD(g) \tag{3}$$

□

4 PAD manipulation procedures.

Just like ROBDDs, the manipulation routines are recursive. Lists of vertices are dynamically created and passed down during recursion. Unlike ROBDDs, however, a mechanism for recognizing the possibility of nondeterminism (*i.e.* creating lists of vertices and returning them up) is provided. Fig. (4) illustrates the pseudocode of a binary operation, the logic OR of two functions.

```
   OR (list op1, list op2)
   {
1  if (terminal case) return (terminal value);
2  opc=op1 ∩ op2; op1=op1 ∪ opc; op2 = op2 ∪ opc;
3  res = comp_lookup(op1, op2);
4  if (res != NULL) return (res ∪ opc);
5  x = top_var(op1, op2);
6  left=OR (op1.x, op2.x); right=OR (op1.x', op2.x');
7  res = pad_find (left, right, x);
8  comp_insert (op1, op2, res);
9  return (res ∪ opc);
   }
```

Figure 4. Pseudocode of `OR()`

The inputs `op1, op2` are lists of vertices. At the top level of recursion, they represent the sets of roots of the two operands. There are two significant departures from conventional BDD-based procedures, in lines (2) and (7), respectively. Line (2) is an immediate application of case (1) in Theorem (4). We seek subfunctions that are common to both operands and remove these subfunctions from the operands. This removal may result in a potentially faster execution, because all the variables in the support of `opc` will not interfere in the computation of $OR(op1 - opc, op2 -$

opc). Similar factorings apply to all other binary operations. The second difference consists of replacing `bdd_find()` with a different procedure, `pad_find()`, in line (7).

```
   pad_find (set left, set right, var x)
   {
1  if (left == right) return (left);
2  if (left == 1) {
3      new_vertex = find_or_create (1,0,x);
4      return (left ∪ {new_vertex});
   }
5  if (right == 1) { /* symmetric case */ }
6  shared = left ∩ right;
7  left = left ∩ shared; right = right ∩ shared;
8  new_vertex = find_or_create (left,right,x);
9  return (shared ∪ {new_vertex});
   }
```

Figure 5. Pseudocode of `pad_find()`

The pseudocode of `pad_find()` is shown in Fig. (5). In ROBDDs, `bdd_find(l, r, x)` is responsible for checking the existence of a BDD node with variable label `x` and with left and right pointers matching `l` and `r`, respectively. In case no such node exists, a new node is allocated and returned. In the case of PADs, two distinct actions are taken by `pad_find()`. Consider the two functions F_l and F_r represented by the sets `left, right`, respectively. The first action consists of checking whether any of F_l, F_r is 1. If (say) F_l is 1, then $DSOD(F) = DSOD(F_l x + F_r x') = DSOD(x + F_r) = \{x\} \cup DSOD(F_r)$. `Pad_find()` (lines 1-4) then augments the list representing F_r with a new element, representing the function x, and returns this new list. This transformation is depicted in Fig. (6), and represents the application of Theorem (5). The second action consists of identifying common terms between `left` and `right`, and factoring them out (lines 6-7). This applies case (3), Theorem (4). This factoring operation is illustrated in Fig. (7).

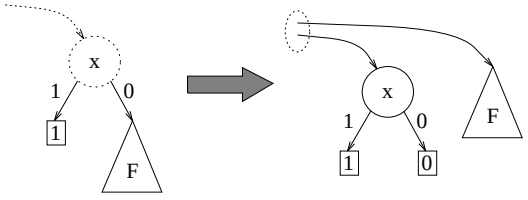

Figure 6. Identification of disjoint-support subfunctions during traversal.

Theorem 6. *Procedure* `OR` *returns a PAD with the disjoint-support property.* □

5 Comparing PADs with ROBDDs.

In this section, we contrast PADs with ROBDDs with respect to two issues, namely the robustness with respect to variable orderings, and the relative performance in terms of memory occupation and CPU time. We compare the two representations on theoretical grounds, and substantiate our findings with experimental results on some benchmark circuits.

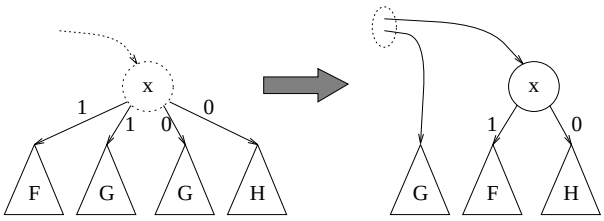

Figure 7. Algebraic reduction of a vertex: a) the vertex prior to reduction; b) after reduction

Since a PAD node size depends on the lists it includes, we carried out comparisons by measuring the actual memory occupation. We assumed bare-bone implementations, in which in particular each ROBDD node takes three machine words. With regards to PAD vertices, we assumed an implementation where each node consists of an array. The first element of the array stores in compact form the number of elements belonging to the left pointer set, the right pointer set, and the variable rank.

We implemented a PAD package and tested it against the Carnegie-Mellon ROBDD package. Time was taken on a HP Vectra VL4 5/133 with 32Mbyte of RAM. We tested the two packages on three series of benchmarks, namely, the IWLS91 combinational multi-level and two-level and the synchronous circuit series. We chose as **initial** variable ordering the variable ordering obtained by the **stable-window-3** reordering algorithm [17]. No variable reordering took place, however, during the execution of any package. We have also tested some ISCAS benchmarks with the ordering provided by Wegener *et al.* in [18].

5.1 Variable orderings.

PADs appear to be less sensitive to variable orderings. There are several theoretical motivations supporting this observation. Consider a function f with a nontrivial DSOD, say , $\{f_1, f_2\}$. The PAD representation will consist of a two-rooted graph, and the total graph size is the sum of the two subgraph sizes. The variable ordering will only affect the size of the subgraphs representing f_1 and f_2.

In the case of ROBDDs, the optimal variable ordering consists of clustering the variables in $S(f_1)$ and $S(f_2)$. The relative order of the two clusters is immaterial. With such an ordering, the total ROBDD size is still the sum of the two subgraph sizes [4]. Any other ordering, interspersing variables from the two supports, may only increase (sometimes substantially) the graph size. In other words, PADs are insensitive with respect to this interspersing.

Example 5. One well known order-sensitive class of functions is given by the formula [1]:

$$f_n = x_1 x_2 + x_3 x_4 + \cdots x_{2n-1} x_{2n} \tag{4}$$

The ROBDD size for this class of functions can range from $2n$ to over 2^n, depending on the ordering. The structure of PADs for f_n is given in Fig. (8). The size is $2n$, regardless of the ordering. □

We now report on how the discovery of a disjoint-support decomposition helped improving substantially ROBDD performance.

Example 6. The synthesis benchmark *pair* has 173 inputs and 137 outputs. The ROBDDs of the complete circuit take, after sift-based dynamic reordering, over 53000 nodes.

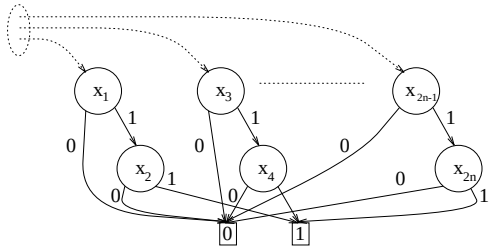

Figure 8. PAD structure for the functions f_n of Example (6).

The most expensive outputs turn out to be $w5$ and $b6$, consisting of 5182 and 9076 nodes, respectively. Using PADs, we found out that both outputs can be expressed as the sum of four disjoint-support functions. The total PAD size was one seventh of the total ROBDD size. For $w5$, the partition was:
$\{n; w2\}$
$\{o; m; z2; y2; e3; d3; a3; c3; b3\}$
$\{t; s; l; p; p1; q1; d2; c2; b2; g2; n2; e2; \}$
$\{f; b; u; q; r; w; v; n1; m1; l1; v0; d1; y0; x0;$
$w0; z0; r1; c1; b1; a1; g1; i1; h1; e1; f1; o1; k1; j1; \}$

For $b6$, we found:
$\{l0; l3; \}$
$\{m0; k0; r3; o3; t3; s3; q3; n3; p3; \}$
$\{r0; q0; j0; n0; g4; i4; f4; d4; e4; q5; p5; p4; \}$
$\{d0; z; s0; o0; p0; u0; t0; v4; r4; z4; y4; x4; w4; u4;$
$t4; s4; c5; e5; d5; b5; m5; l5; n5; r5; a5; g5; o5; i5; j5; k5; \}$

Notice that the supports of $w5$ and $b6$ are disjoint as well. It is thus possible to obtain a good order for both functions, by clustering together the variables of the eight groups. We tested this order on ROBDDs, and were able to reduce the memory occupation of $w5$ and $b6$ to 118 and 362 nodes, respectively. The total memory occupation for *pair* was reduced to 15319 nodes. □

Even when a function cannot be expressed by a nontrivial DSOD, the flexibility of PADs with respect to orderings may be useful; for instance if ordering requirements for multiple ROBDDs conflict. This conflict is less likely in the case of PADs. We now substantiate this observation:

Example 7. Wegener *et al.* provide in [18] the best ordering known for several ISCAS benchmarks. We analyzed the benchmark C880. Three outputs (878GAT.442, 879GAT.441, 880GAT.440) result in the largest ROBDDs.

All of these outputs but 878GAT.442 can be decomposed into the sum of a 2-variable function and a larger one. The small functions depend on (210GAT.49, 91GAT.23) and (210GAT.49, 96GAT.24). The function 879GAT.441 then suggests modifying Wegener's order by moving 210GAT.49 and 91GAT.23 up top or at the bottom. Other similar moves are suggested by 880GAT.440. In trying these orders, we found out that, while we were indeed reducing the ROBDD of a single function, we were increasing the size of the other functions. We also tried moving only 210GAT.49. In no case we could beat Wegener's order. Using PADs with Wegener's order, instead, we reduced the memory occupation by about 50%. □

5.2 Memory, CPU time, and complementation.

We tried two different versions of ROBDDs, without and woth complement edges respectively. The reason for testing both cases is that the current implementation of PADs does not support constant-time / constant-space complementation.

Table (1) reports the results in terms of nodes (`nod`), machine words (`mem`) and CPU time (`CPU`) for the two ROBDD versions against PADs. PADs turn out to be more compact almost always when compared with complement-free ROBDDs, with some penalty in CPU time. ROBDDs with complement edges, however, are often more efficient than PADs on both fronts. Empirically, we found the following three reasons:

- Complementations in PADs do, so far, have a cost, in terms of memory and CPU time. If a function f has a significant DSOD, then it is easy to verify that its complement will not, and the PAD of f' will be "close" to a ROBDD. Moreover, the PAD of f' inherits all the inefficiencies of a poor ordered ROBDD.
- Data inserted and looked up in `computed_table` are arrays. The hash function depends on all array elements, and its computation is consequently more costly.
- Some terminal cases (like in `OR(f,f')`) are not identified by PAD routines. These inefficiencies (space and time) manifest themselves mostly in arithmetic-type circuits. PADs maintain an edge over ROBDDs in the case of synchronous benchmark circuits.

To this regard, we wish to point out that we are already aware of ways to circumvent the complementation issue, and that a constant-time complementation is in fact possible for PADs. We are currently working in this direction.

6 Conclusions and future work.

In this paper, we inspected the possibility that more compact and flexible graph-based representations may be obtained by relaxing some of the constraints imposed on ROBDDs. We considered allowing more output edges from a vertex. This is conceptually equivalent to considering a NFA-based representation, as opposed to a DFA-based style. We have suitably restricted the type of nondeterminism, however, so that the graph representation of a function is still unique.

Compared with ROBDDs, we have shown that the new representation is more robust with respect to poor variable ordering, in no case less compact, but quite often more compact.

Currently, the complement f' of a function f has a separate representation, and it is difficult to detect when two functions are each other's complement. This has a performance penalty in terms of memory and CPU time, which occurs mostly in arithmetic-type circuits. We are already aware of ways for circumventing the complementation problem, and are working in this direction. We also expect benefits from a dynamic reordering heuristic tailored especially for PADs, and from the use of more general decomposition styles.

Benchmark	ROBDD					PAD			RATIOS			
	w/o c.edges		w c.edges						w/o c.edges		w c.edges	
	nod	mem	nod	mem	CPU	nod	mem	CPU	nod	mem	nod	mem
MULTILEVEL												
C1355.iscas	43140	129420	39458	118374	4.38	42037	132099	10.73	2.6%	-2.0%	-6.1%	-10.4%
C1908.iscas	31960	95880	22771	68313	2.82	30223	114388	8.29	5.7%	-16.2%	-24.7%	-40.3%
C432.iscas	1314	3942	1229	3687	0.11	1144	3673	0.42	14.9%	7.3%	7.4%	0.4%
C499.iscas	42890	128670	39377	118131	3.18	42003	130725	5.64	2.1%	-1.6%	-6.3%	-9.6%
C880.iscas	9888	29664	9864	29592	0.47	6467	20962	1.64	52.9%	41.5%	52.5%	41.2%
cht	151	453	150	450	0.01	149	353	0.01	1.3%	28.3%	0.7%	27.5%
DES	104696	314088	65840	197520	5.81	67807	192298	10.97	54.4%	63.3%	-2.9%	2.7%
frg1	171	513	170	510	0.04	147	375	0.10	16.3%	36.8%	15.6%	36.0%
pair	53205	159615	52449	157347	1.35	6191	19259	3.77	759.4%	728.8%	747.2%	717.0%
sct	124	372	105	315	0.01	81	221	0.03	53.1%	68.3%	29.6%	42.5%
x1	1212	3636	997	2991	0.12	722	2003	0.28	67.9%	81.5%	38.1%	49.3%
x4	756	2268	732	2196	0.08	630	1689	0.14	20.0%	34.3%	16.2%	30.0%
TWOLEVEL												
apex1.pla	1607	4821	1579	4737	0.23	1505	4293	0.60	6.8%	12.3%	4.9%	10.3%
apex5.pla	2482	7446	2450	7350	0.98	2099	5889	2.02	18.2%	26.4%	16.7%	24.8%
duke2.pla	440	1320	434	1302	0.06	437	1079	0.15	0.7%	22.3%	-0.7%	20.7%
e64.pla	633	1899	631	1893	0.10	632	1263	0.18	0.2%	50.4%	-0.2%	49.9%
misex2.pla	105	315	103	309	0.00	104	228	0.02	1.0%	38.2%	-1.0%	35.5%
FSM												
ex2	326	978	324	972	0.02	325	672	0.04	0.3%	45.5%	-0.3%	44.6%
ex3	115	345	113	339	0.01	114	242	0.01	0.9%	42.6%	-0.9%	40.1%
ex7	123	369	121	363	0.02	122	259	0.02	0.8%	42.5%	-0.8%	40.2%
s444	262	786	234	702	0.04	222	604	0.06	18.0%	30.1%	5.4%	16.2%
s641	1123	3369	1007	3021	0.09	807	2215	0.17	39.2%	52.1%	24.8%	36.4%
s713	1157	3471	1029	3087	0.12	704	1933	0.17	64.3%	79.6%	46.2%	59.7%
WEG.ORDER												
C1355.iscas	27869	83607	25866	77598	2.46	27476	84103	6.65	1.4%	-0.6%	-5.9%	-7.7%
C1908.iscas	7542	22626	5526	16578	0.64	7263	23521	1.86	3.8%	-3.8%	-23.9%	-29.5%
C432.iscas	1146	3438	1087	3261	0.13	1036	3587	0.44	10.6%	-4.2%	4.9%	-9.1%
C499.iscas	27869	83607	25866	77598	1.76	27476	84103	3.06	1.4%	-0.6%	-5.9%	-7.7%
C880.iscas	4073	12219	4053	12159	0.20	2894	9539	0.75	40.7%	28.1%	40.0%	27.5%

Table 1. ROBDD vs. PAD in size and performance

References

[1] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. on Computers*, 35(8):677–691, August 1986.

[2] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a BDD package. In *Proc. DAC*, pages 40–45, June 1990.

[3] O. Coudert and J.C. Madre. A unified framework for the formal verification of sequential circuits. In *Proc. ICCAD*, pages 126–129, November 1990.

[4] S. Malik, A. R. Wang, R. K. Brayton, and A. Sangiovanni-Vincentelli. Logic verification using binary decision diagrams in a logic synthesis environment. In *Proc. ICCAD*, pages 6–9, November 1988.

[5] Y. Matsunaga and M. Fujita. Multi-level logic optimization using binary decision diagrams. In *Proc. ICCAD*, pages 556–559, November 1989.

[6] H. Touati, H. Savoj, B. Lin, R.K. Brayton, and A. Sangiovanni-Vincentelli. Implicit state enumeration of finite state machines using BDD's. In *Proc. ICCAD*, pages 130–133, November 1990.

[7] F. Corno, P. Prinetto, and M. Sonza Reorda. Using symbolic techniques to find the maximum clique in very large sparse graphs. In *Proc. EDAC*, pages 320–324, March 1995.

[8] Y-T. Lai and S. Sastry. Edge-valued binary decision diagrams for multi-level hierarchical verification. In *Proc. DAC*, pages 240–243, June 1992.

[9] R. E. Bryant. On the complexity of vlsi implementations and graph representations of boolean functions with application to integer multiplication. *IEEE Trans. on Computers*, 40:205–213, 1991.

[10] I. Wegener. *The complexity of Boolean Functions*. John Wiley and Sons, 1987.

[11] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proc. ICCAD*, pages 42–47, November 1993.

[12] S. J. Friedman and K. J. Supowit. Finding the optimal variable ordering for binary decision diagrams. *IEEE Trans. on Computers*, 39:710–713, 1990.

[13] R. Drechsler, A. Sarabi, M. Theobald, B. Becker, and M. A. Perkowski. Efficient representation and manipulation of of switching functions based on ordered kronecker functional decision diagrams. In *Proc. DAC*, pages 415–419, June 1994.

[14] J. Jain, M. Abadir, J. Bitner, D. Fussell, and J. Abraham. Ibdds: An efficient functional representation for digital circuits. In *Proc. DAC*, pages 441–446, June 1992.

[15] H. A. Curtis. *A new approach to the design of switching circuits*. Van Nostrand, Princeton, N.J., 1962.

[16] J. P. Roth and R. M. Karp. Minimization over boolean graphs. *IBM Journal*, pages 661–664, April 1962.

[17] M. Fujita, Y. Matsunaga, and T. Kakuda. On variable ordering of binary decision diagrams for the application of multi-level logic synthesis. In *Proceedings of the European Conference on Design Automation*, pages 50–54, 1991.

[18] B. Bollig, M. Lobbing, and I. Wegener. Simulated annealing to improve variable orderings for obdds. In *International Workshop on Logic Synthesis*, page 5.1. IEEE-ACM, 1995.