[3] S. Devadas and K. Keutzer, “Synthesis of robust delay-fault-testable circuits: Theory,” *IEEE Trans. Computer-Aided Design*, vol. 11, pp. 87–101, Jan. 1992.
[4] ——, “Synthesis of robust delay-fault-testable circuits: Practice,” *IEEE Trans. Computer-Aided Design*, vol. 11, pp. 227–300, Mar. 1992.
[5] I. Pomeranz and S. M. Reddy, “Achieving complete delay fault testability by extra inputs,” in *Proc. Int. Test Conf.*, 1991, pp. 273–282.
[6] V. A. Vardanian, “On completely robust path delay fault testable realization of logic functions,” in *Proc. VLSI Test Symp.*, 1996, pp. 302–307.
[7] N. K. Jha, I. Pomeranz, S. M. Reddy, and R. J. Miller, “Synthesis of multi-level combinational circuits for complete robust path delay fault testability,” in *Proc. FTCS*, 1992, pp. 280–287.
[8] A. K. Pramanick and S. M. Reddy, “On the design of path delay fault testable combinational circuits,” in *Proc. FTCS*, 1990, pp. 374–381.
[9] W. Ke and P. R. Menon, “Delay-testable implementations of symmetric functions,” *IEEE Trans. Computer-Aided Design*, vol. 14, pp. 772–775, June 1995.
[10] I. Pomeranz and S. M. Reddy, “On synthesis-for-testability of combinational logic circuits,” in *Proc. ACM/IEEE Design Automation Conf.*, 1995, pp. 126–132.
[11] Z. Kohavi, *Switching and Finite Automata Theory*. New York: McGraw Hill, 1977.
[12] D. L. Dietmeyer, “Generating minimal covers of symmetric function,” *IEEE Trans. Computer-Aided Design*, vol. 12, pp. 710–713, May 1993.
[13] C. Ding, G. Xiao, and W. Shan, *The Stability Theory of Stream Ciphers*. Berlin, Germany: Springer-Verlag, 1991. LNCS.
[14] Y. X. Yang and B. Guo, “Further enumerating Boolean functions of cryptographic significance,” *J. Cryptol.*, vol. 8, no. 3, pp. 115–122, 1995.
[15] S. Chakrabarti, “Studies in redundancy and testable design of logic circuits,” Ph.D. thesis, Univ. Calcutta, Calcutta, India, 1998.

New Multivalued Functional Decomposition Algorithms Based on MDDs

Craig M. Files and Marek A. Perkowski

***Abstract*—This paper presents two new functional decomposition partitioning algorithms that use multivalued decision diagrams (MDDs). MDDs are an exceptionally good representation for generalized decomposition because they are canonical and they can represent very large functions. Algorithms developed in this paper are for Boolean/multivalued input and output, completely/incompletely specified functions with application to logic synthesis, machine learning, data mining and knowledge discovery in databases. We compare the run-times and decision diagram sizes of our algorithms to existing decomposition partitioning algorithms based on decision diagrams. The comparisons show that our algorithms are faster and do not result in exponential diagram sizes when decomposing functions with small bound sets.**

***Index Terms*—Algorithms, logic design, unsupervised learning.**

I. Introduction

Functional decomposition is known as expressing a function as a composition of two or more functions. While many papers were written about the topic of functional decomposition there was no comprehensive approach until Ashenhurst presented a unified theory of functional decomposition, and for the first time defined its basic properties in [1], [2]. Curtis used the theorems of Ashenhurst to develop a generalized form of decomposition in [3] and [4]. There have been many other proposed types of functional decompositions since the advent of Curtis decomposition. But, the fundamentals of Ashenhurst and Curtis decomposition provide essential insight into a wide range of functional decomposition types.

Two new functional decomposition partitioning algorithms that are based on multivalued decision diagrams (MDDs) [5] are presented in this paper: PARTITION and EVAL. Both algorithms are compared to the existing *cut_level* and LPV (Lai, Pedram, and Vrudhula) binary decision diagram (BDD)-based functional decomposition partitioning algorithms developed by Lai, Pedram, and Vrudhula [6]–[8]. The *cut_level* algorithm is very well known, but is based on reordering the variables in the decision diagram. This can be a problem because variable reordering may lead to decision diagrams of exponential size [9]. This is the advantage of the PARTITION algorithm over the *cut_level* algorithm because the PARTITION algorithm does not reorder variables in the decision diagram. The LPV algorithm is much faster than the *cut_level* algorithm and can quickly evaluate many partitions, but the LPV algorithm can only be used to determine *column multiplicities*, to decompose a function the *cut_level* algorithm must be used.

The advantage of the EVAL algorithm over the LPV algorithm is based on the way the two algorithms construct partition tables and determine column equalities. The LPV algorithm constructs a partition table by constructing each column in the table. After the partition table is created each pair of columns is checked for equality. For completely and incompletely specified functions, two columns are equal if their encoded integer values are equal. An extension to the LPV algorithm for incompletely specified functions is presented in this paper to find columns that are compatible (by setting output don’t cares in the columns, the two columns can be made equal).

The EVAL algorithm constructs the partition table by rows and checks if two columns are equal while constructing the partition table. This makes it possible to determine that two columns are not equal or not compatible before completing their construction. Of course, if the two columns are equal then the two columns must be fully constructed. The advantage of the EVAL algorithm is that after the construction of the partition table no extra computation is needed. The LPV algorithm must construct the partition table and then determine column equalities and column compatibilities.

Section II gives the general notations for functional decomposition. Section III presents the Lai, Pedram, and Vrudhula algorithms and our new MDD-based algorithms. Section IV shows the experimental results of the algorithms and the paper is concluded in Section V.

II. General Decomposition

The decomposition of a function can be an expression of the function in terms of a composition of other functions. For example, if $f(x_0, x_1, x_2, x_3) = F(\Phi(x_0, x_1), x_2, x_3)$, then the term on the right is a decomposed function that is equivalent in behavior to the original function f.

In general, an n-input, single output Boolean function, f: $\{0, 1\}^n \rightarrow \{0, 1\}$, has the set of input variables $X = \{x_0, x_1, \ldots, x_{n-1}\}$. The number of variables in set X is denoted by $|X|$.

Definition 1: Let $A \subset X$ and $B \subset X$, where $A \neq \emptyset$ and $B \neq \emptyset$. A **partition**, denoted as $A|B$, exists if $A \cap B = \emptyset$ and $A \cup B = X$.

Definition 2: A function $f(x_0, x_1, \ldots, x_{n-1})$ has an **Ashenhurst simple disjunctive decomposition** [1], if f can be represented by $f = F(\Phi(B), A)$. This is known as partitioning the input variables into the **bound set** B and the **free set** A. The variables in B and A are known as **bound variables** and **free variables**, respectively.

Manuscript received May 12, 1999; revised April 8, 2000. This paper was recommended by Associate Editor L. Stok.

The authors are with The Department of Electrical Engineering, Portland State University, Portland, OR 97207-0751 USA (e-mail: cfiles@ee.pdx.edu).

Publisher Item Identifier S 0278-0070(00)07476-5.

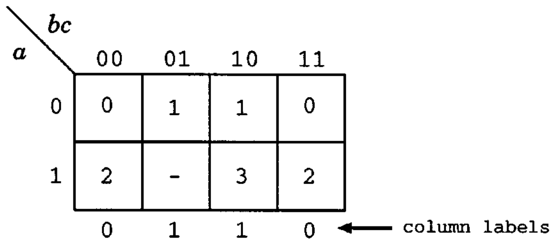


Fig. 1. Partition matrix $\{a\}|\{b, c\}$.

Definition 3: For the partition $A|B$ on X, a **partition matrix** representation of $f(X)$ is defined as a rectangular array of the 2^n functional values of f, arranged in $2^{|A|}$ rows and $2^{|B|}$ columns.

Definition 4: The number of distinct column vectors in a partition matrix is called the **column multiplicity** of the partition and is denoted by ν.

Theorem 1: A function $f(x_0, x_1, \ldots, x_{n-1})$ has an Ashenhurst simple disjunctive decomposition, denoted by $f = F(\Phi(B), A)$ if the partition matrix $A|B$ has column multiplicity $\nu \leq 2$.

Proof: Proof is given in [4]. ■

Obviously, Theorem 1 can be expanded for partitions that have larger column multiplicities.

Definition 5: A Boolean function $f(x_0, x_1, \ldots, x_{n-1})$ has a **Curtis disjunctive decomposition** [10], denoted by the composite function

$$f = F(\Phi_0(B), \Phi_1(B), \ldots, \Phi_{k-1}(B), A)$$

if the partition matrix $A|B$ has column multiplicity $\nu \leq 2^k$, where $k < |B|$.

A multivalued function over multivalued arguments x_i, $0 \leq i < n$, denoted by $f(x_0, x_1, \ldots, x_{n-1})$, takes output values from a finite set of values. A multivalued variable x_i can take values from the set $Q_i = \{q_0, q_1, \ldots, q_{|Q_i|-1}\}$. For reference, Q_i denotes the set of values that variable x_i can have. Each symbolic value q_k can be associated with a unique integer k, and because only integer values are considered, then $Q_i = \{0, 1, \ldots, |Q_i| - 1\}$.

A multivalued function f is a function which maps vertices in $Q_0 \times Q_1 \times \cdots \times Q_{n-1}$ to Q_f, formally, f: $Q_0 \times Q_1 \times \cdots \times Q_{n-1} \rightarrow Q_f$.

Definition 6: For the partition $A|B$ on X, a **multivalued partition matrix** representation of $f(X)$ is defined as a rectangular array of the functional values of f. Given that $A = \{x_j, \ldots, x_k\}$ and $B = \{x_l, \ldots, x_m\}$, the partition matrix is arranged in $|Q_A| = |Q_j| \times \cdots \times |Q_k|$ rows and $|Q_B| = |Q_l| \times \cdots \times |Q_m|$ columns.

The definitions of simple disjunctive decompositions can easily be applied to functions that are incompletely specified. An incompletely specified function can be represented in a partition matrix, where the column multiplicity is found by finding columns that are *compatible*.

Definition 7: Two columns in a partition matrix are **compatible** if for every row, the output values of both columns are equal, or if at least one of the output values is a *don't care*.

Example 1: Given the partition $\{a\}|\{b, c\}$ of f in Fig. 1, the columns "00" and "11" are equal, and columns "01" and "10" are compatible. By setting the don't care in column "01" to 3, columns "01" and "10" are now equal and the column multiplicity $\nu(bc|a) = 2$. Because $\nu = 2$ and the number of Φ composite functions is $k = \lceil \log_2 \nu \rceil = \lceil \log_2 2 \rceil = 1$, f can be written as $F(\Phi(b, c), a)$.

III. Decision Diagram Based Decomposition

Because partition matrices are exponential in the number of functional inputs, many researchers have looked for different data structures to represent partition matrices, including decision diagrams. This section discusses decision diagram-based functional decomposition [11], [12], [6]–[8].

Definition 8: A **decision diagram** over a set of variables X and a nonempty terminal set T is a connected, directed acyclic graph $G = (V, E)$ with the following properties:

- a vertex $v_i \in V$ is either a *nonterminal* or a *terminal vertex*;
- each nonterminal vertex v_i represents a variable $x_i \in X$ and v_i has exactly $|Q_i|$ **successors** in V, given that x_i has $|Q_i|$ cofactors;
- each terminal vertex v_i has no successors and is labeled with a value $t_i \in T$ where $T = Q_f$, given that Q_f is the set of output values function f can have;
- a **root vertex** is the top vertex in G, i.e., no vertex in the graph has a root vertex as a successor.

Definition 9: A **decision diagram partition matrix** of a function can be represented by partitioning the input variables in the decision diagram, such that all of the bound variables are above (at the top of the graph) all of the free variables. Note, the order of variables within the bound set or within the free set has no effect on the partition found.

Definition 10: The **cut_level** designates the boundary between the bound variables and the free variables in the decision diagram.

The method for detecting decompositions using the decision diagram canonical form is called the *cut_level* algorithm [8]. The algorithm states that the number of distinct columns in a decision diagram partition matrix is the number of vertices below the cut_level that have an edge that crosses the cut_level. This statement is true because of the canonical representation of the decision diagram. If a function is completely specified, then the column multiplicity is just the number of distinct columns found in the decision diagram.

As proposed in [11], the *cut_level* algorithm can easily be extended to multivalued functions. The following example illustrates the *cut_level* algorithm for multivalued functions.

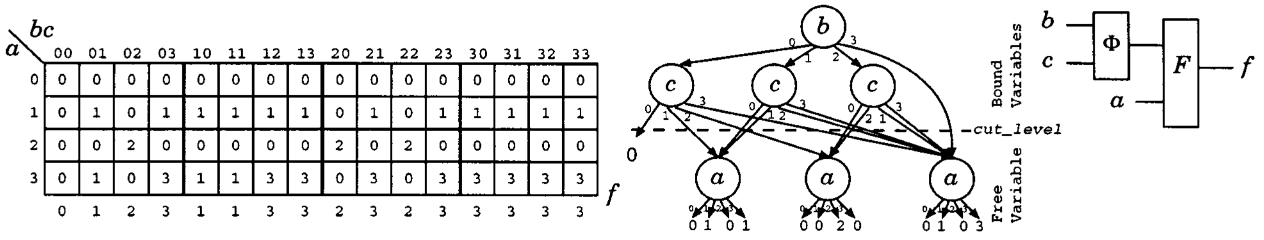
Example 2: Consider the four-valued function in Fig. 2, where the partition $\{a\}|\{b, c\}$ of the function $f(a, b, c)$ is shown in Fig. 2(a). The column multiplicity is the number of vertices below the *cut_level* that have an edge that crosses the *cut_level*. From the decision diagram shown in Fig. 2(b) the column multiplicity is four, which implies the decomposition $f = F(\Phi(b, c), a)$, where Φ is a four-valued function. Fig. 2(c) illustrates the decomposed functional blocks.

The *cut_level* algorithm, although popularly used, does have its problems. To evaluate partitions, the variables must be reordered within the decision diagram package. The process of variable reordering can actually lead to diagrams that have exponential size [9], which can result in increased run-times. The remaining algorithms presented in this paper do not reorder the variables in a decision diagram.

A. Bound Set Evaluation without Reordering

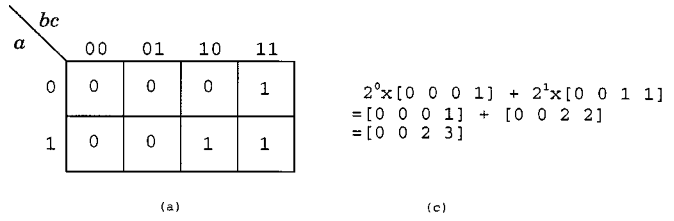
Given that variable reordering can cause large timing and size constraints, researchers looked for a method that would require very little reordering. The only algorithm that has been published, as far as we know, is the LPV evaluation algorithm proposed by Lai, Pedram, and Vrudhula [8]. An advantage of the LPV algorithm is that the inclusion or exclusion of bound variables is done without reordering the variables in the BDD package. The rest of this subsection discusses the implementation of the LPV algorithm. At the end of this subsection we extend the LPV algorithm for multivalued functions.

The LPV algorithm determines the number of distinct columns by using *bit vectors* to represent each of the columns. A **bit vector** is defined as a set of Boolean values corresponding to the functional output values of a function in order. Because each column in a partition matrix may be represented by a bit vector. By encoding a bit vector to



a \ bc	00	01	02	03	10	11	12	13	20	21	22	23	30	31	32	33
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	0	1	1	1	1	1	0	1	0	1	1	1	1	1
2	0	0	2	0	0	0	0	0	2	0	2	0	0	0	0	0
3	0	1	0	3	1	1	3	3	0	3	0	3	3	3	3	3
f	0	1	2	3	1	1	3	3	2	3	2	3	3	3	3	3

Fig. 2. Example of MDD-based decomposition. (a) Partition matrix $\{a\}|\{b, c\}$. (b) Corresponding MDD. (c) Decomposed blocks.



a \ bc	00	01	10	11
0	0	0	0	1
1	0	0	1	1

```
2⁰x[0 0 0 1] + 2¹x[0 0 1 1]
=[0 0 0 1] + [0 0 2 2]
=[0 0 2 3]
```

Fig. 3. LPV example. (a) Partition $\{a\}|\{b, c\}$. (b) LPV encoding.

an integer value, the determination of column equality is done by comparing the encoded integer value of each column. If the encoded integer values of two columns are equal, then the two columns are equal. Because the integers can be very large (the size of each bit vector is exponential in the number of free variables), an edge-valued BDD is used to represent each of the encoded integer values.

Example 3: Given is the partition matrix $\{a\}|\{b, c\}$ for $f(a, b, c)$ shown in Fig. 3(a). The first column in the partition can be encoded as $2^0 \times 0 + 2^1 \times 0$, where the encoding is given by the multiplication of $2^{\text{rownumber}}$. This operation can be continued until all columns are found, which is the same as finding the integer vectors for each row, namely $[0, 0, 0, 1]$ and $[0, 0, 1, 1]$, and encoding them such that $2^0 \times [0, 0, 0, 1] + 2^1 \times [0, 0, 1, 1]$ as shown in Fig. 3(b). The column multiplicity is three, found by the number of different encoded integer values in the vector $[0, 0, 2, 3]$.

The following is a possible extension of the LPV algorithm to incompletely specified functions (this is not presented in [8]). Instead of using one encoded integer value to represent a column, a pair of encoded integer values are used. The pair of encoded integers are the minimum and maximum values of a given column bit vector, denoted as [min, max]. The minimum and maximum encoded integer values are obtained by forcing all *don't cares* in the bit vector to 0 and 1, respectively. For instance, given the bit vector of a column as $00-10-$, where a dash denotes a *don't care* value, the minimum and maximum values are 4 and 13. The binary representations of 4 and 13 are 000 100 and 001 101. If a column does not have *don't cares* then $\min = \max$.

The pair [min, max] can be used to compute the corresponding cube C that represents the output values of a column. Two columns are compatible if their corresponding cubes C_1 and C_2 have a nonempty intersection. Converting the integer values to cubes and checking if the two cubes intersect are linear procedures. The problem is that the cubes are exponential in the number of free variables. Because of the edge-valued BDD representation of the encoded integer values, computation of the cubes is generally less than exponential, but can still be quite large.

To compare our new algorithm EVAL (presented in the next subsection) with the LPV algorithm, we have created an LPV algorithm for multivalued functions. This extension is exactly the same as in the LPV algorithm for incompletely specified functions except that it uses powers of Q_f instead of powers of two to encode each row, given that the functional output takes values from the set Q_f.

B. New Decomposition Algorithms Based on MDDs

The decomposition strategy proposed here is the same as the strategy of Lai, Pedram, and Vrudhula algorithms, where one algorithm (EVAL) is used to evaluate the column multiplicity of a partition and a second algorithm (PARTITION) is used to decompose the function. Two algorithms are used because the algorithm to determine column multiplicity is much faster than the algorithm for decomposing a partition. Both algorithms were developed for large Boolean/multivalued, completely/incompletely specified functions to check small bound sets.

Our first algorithm, called EVAL determines column incompatibilities while traversing a functional graph by using an incompatibility graph to store the incompatibilities between columns [13]. While traversing a functional graph, the EVAL algorithm extracts sets of partition matrix rows. Each set of rows is a subset of all rows in a partition matrix, and thus, pairs of columns can be checked for incompatibilities over the subset of rows. When the EVAL algorithm completes its traversal, the columns have been checked for incompatibilities over all rows of a partition matrix. Incompatibility of two columns is determined by simultaneously traversing the graphs that represent the columns over a given set of rows. If the traversals reach terminal vertices that have different values and neither vertex is a *don't care,* then the columns are incompatible as defined in Definition 7.

Before running the EVAL algorithm, an incompatibility graph is created with no edges and each vertex in the graph is associated with a column in the partition matrix. If two columns are found to be incompatible over a given set of partition matrix rows, an edge is placed in the incompatibility graph between the two vertices that represent the two columns. If two columns are found to be incompatible during the traversal, the columns are not checked for incompatibility over any remaining set of rows. In the worst case (two columns are compatible), columns are checked for incompatibilities over all possible row values. In the best case, the columns are found to be incompatible for the first row in the partition matrix, and only one row is evaluated.

The EVAL algorithm extracts a set of rows from the decision diagram by using a *depth-first* algorithm that simultaneously traverses subgraphs of the graph representation of f. The number of subgraphs traversed equals the number of columns, $|Q_B|$, in the partition matrix. The algorithm starts by traversing the functional diagram of f, until a vertex that represents a bound variable, x_i, is reached. The traversal is continued by simultaneously traversing each of the $|Q_i|$ cofactors of the vertex. When the traversals reach vertices that represent a second bound variable, x_j, the traversal is continued by simultaneously traversing each of the $|Q_j|$ cofactors of each vertex. At this point the number of subgraphs being traversed simultaneously is $|Q_i| \times |Q_j|$. The traversal is continued until the number of subgraph traversals is $|Q_B|$. Each of the $|Q_B|$ subgraphs represents a column over a set of rows, and therefore, pairs of subgraphs are checked for incompatibilities. The traversal routine then continues to find the next set of rows. The EVAL pseudo code is shown in Algorithm 1.

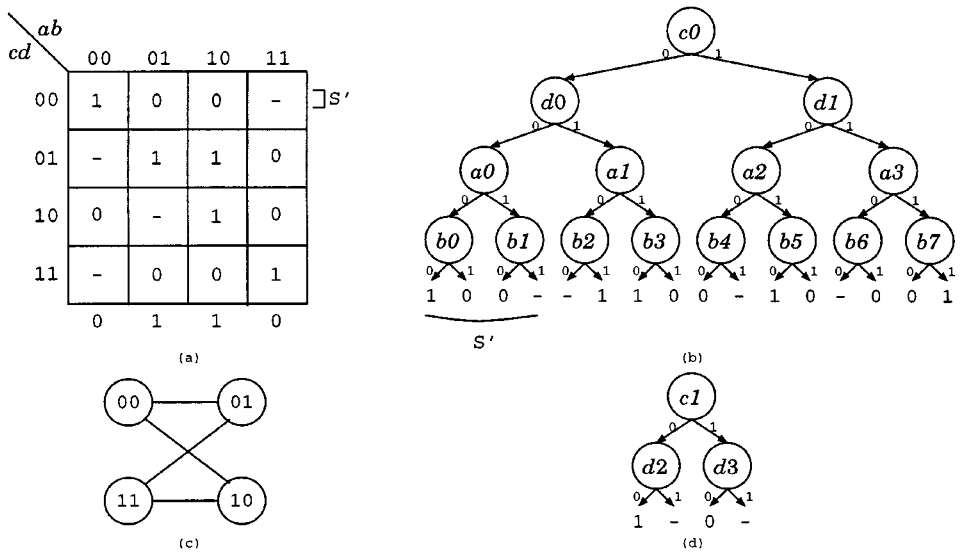


Fig. 4. EVAL and PARTITION example. (a) Partition matrix $\{c, d\}|\{a, b\}$. (b) Decision tree. (c) EVAL incompatibility graph. (d) 00 column returned by PARTITION.

```
Algorithm 1 EVAL pseudo code.
EVAL(vector of MDD vertices: $\underline{S}$)
  let $x$ be the variable represented by the
  set of vertices in $\underline{S}$
  let $|Q_x|$ be the number of cofactors of
  variable $x$
  let $\underline{S}_i$ be a vector of vertices repre-
  senting the $i$th cofactor of each vertex
  in $\underline{S}$
  let $[\underline{S}_j, \underline{S}_k]$ be a vector of vertices such
  that $\underline{S}_j$ and $\underline{S}_k$ are subsets of the vector
  if $x$ is a bound variable
    call EVAL($[\underline{S}_0, \underline{S}_1, \ldots, \underline{S}_{|Q_x|-1}]$)
  else $x$ is a free variable
    if $|S| = |Q_B|$
      $\underline{S}$ denotes all the columns in the
      partition matrix for all pairs
      $\{s_i, s_j\} \in \underline{S}$
        if there does not exist an edge in
        the incompatibility graph between
        $s_i$ and $s_j$
          if elements $\{s_i, s_j\}$ are incompat-
          ible columns
            add an edge in the incompati-
            bility graph between $s_i$ and $s_j$
      else
        for $i = 0$ to $|Q_x| - 1$
          call EVAL($\underline{S}_i$)
```

Example 4: Given the function $f(a, b, c, d)$ and the partition $\{c, d\}|\{a, b\}$ shown in Fig. 4(a), the corresponding decision tree in Fig. 4(b) is used to explain the EVAL algorithm. First, an incompatibility graph is created with four vertices and no edges, the vertices represent each of the columns $\{00, 01, 10, 11\}$ in the partition matrix and the edges between vertices represent incompatibilities between columns.

The EVAL algorithm is started by traversing the decision tree representation of f. The root vertex, $c0$, does not represent a bound variable, so EVAL is called with the 0-cofactor of $c0$: $[d0]$. d is not in the bound set, so EVAL is called with the 0-cofactor of $d0$: $[a0]$. Variable a is in the bound set, so EVAL is called with both cofactors of $a0$: $[b0, b1]$. Variable b is in the bound set, so EVAL is called with all four cofactors of $[b0, b1]$: $[1, 0, 0, -]$ (shown in the figure as S'). The number of vertices in vector S is the same as the number of columns in the partition, so the vertices in vector S are checked for incompatibility. Edges are placed in the incompatibility graph between the pairs of columns $\{00, 01\}$ and $\{00, 10\}$ because the first element in S is incompatible with the second and third elements in S. The algorithm then moves back to the vertex $d0$, and then traverses the 1-cofactor of $d0$: $[a1]$. Because variables a and b are bound variables, EVAL is iteratively called with both cofactors of $a1$: $[b2, b3]$, and then all four cofactors of $[b2, b3]$: $S = [-, 1, 1, 0]$. Edges are placed between the pairs of columns $\{01, 11\}$ and $\{10, 11\}$ in the incompatibility graph [shown in Fig. 4(c)]. By traversing the remaining vertices in the diagram, the vectors $[0, -, 1, 0]$ and $[-, 0, 0, 1]$ are checked for incompatibility. Because no additional incompatibilities exist, the incompatibility graph is not changed. After EVAL traverses all vertices in the diagram, graph coloring of the incompatibility graph determines that the column multiplicity of this partition is two. Notice that $\nu = 2$ is also true for the partition matrix shown in Fig. 4(a).

To decompose a function given a partition, the PARTITION algorithm is used. The PARTITION algorithm returns decision diagrams that represent each of the columns in the partition matrix. Algorithm 2 shows the recursive procedure that returns the left-most column in the partition matrix, by only traversing the 0-cofactor of each bound variable. The full algorithm of PARTITION simultaneously calculates all columns in the partition matrix

```
Algorithm 2 PARTITION pseudo code to find
the left-most column.
PARTITION($v$: vertex)
  if $v$ is a terminal vertex
    return $v$
  let $x$ be the variable represented by $v$
  let $|Q_x|$ be the number of cofactors of
  variable $x$
  let $v_i$ be the $i$th cofactor of the vertex
  $v$, $0 \le i < |Q_x|$
  if $x$ is a bound variable
    return PARTITION ($v_0$)
```

TABLE I
RUN-TIME COMPARISONS OF THE EVAL AND LPV ALGORITHMS

	EVAL				LPV			
filename	2	4	6	8	2	4	6	8
9sym	0.1	1.0	10.0	94.0	3.0	3.0	8.0	49.0
alu2	0.1	3.0	33.0	285.0	10.0	13.0	25.0	124.0
b12	1.0	3.0	21.0	288.0	713.0	285.0	180.0	499.0
frg1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
sao	1.0	3.0	17.0	159.0	9.0	10.0	15.0	35.0
spla	5.0	20.0	104.0	>1000	>1000	>1000	1106	>1000
t481	0.1	3.0	46.0	514.0	>1000	>1000	877.0	>1000
table3	4.0	28.0	173.0	882.0	384.0	237.0	212.0	431.0
pal2	0.1	0.1	2.0	-	1.0	1.0	3.0	-
c1b	0.1	0.1	1.0	-	1.0	1.0	2.0	-
c2b	1.0	7.0	153	>1000	>1000	>1000	>1000	>1000
c3b	2.0	10.0	279	>1000	>1000	>1000	>1000	>1000
d1	0.1	1.0	9.0	-	>1000	>1000	>1000	-
d2	3.0	24.0	>1000	>1000	>1000	>1000	>1000	>1000
d3	4.0	31.0	>1000	>1000	>1000	>1000	>1000	>1000

TABLE II
RUN-TIME AND SIZE COMPARISONS OF THE PARTITION AND *cut_level* ALGORITHMS

	init.	PARTITION		*cut_level*		$\frac{cut\_level}{PARTITION}$
filename	size	time	size	time	size	size%
9sym	41	0.1	70	0.1	70	100.0
alu2	264	2.0	517	2.0	456	88.3
b12	100	3.0	161	6.0	115	71.5
frg1	230	25.0	415	441.0	19411	4677.4
sao	164	1.0	311	1.0	305	98.1
spla	630	15.0	1637	22.0	1331	81.4
t481	47	3.0	134	7.0	154	115.0
table3	951	15.0	2075	24.0	2297	110.7
pal2	26	0.1	42	0.1	34	81.0
c1b	21	0.1	32	0.1	32	100.0
c2b	100	2.0	182	2.0	182	100.0
c3b	133	4.0	245	7.0	245	100.0
d1	31	0.1	51	0.1	51	100.0
d2	152	2.0	291	2.0	287	98.7
d3	217	8.0	419	8.0	415	99.1

```
else $x$ is a free variable
  return a new vertex with cofac-
  tors [PARTITION ($v_0$), PARTITION
  ($v_1$), ...,PARTITION($v_{|Q_x|-1}$)].
```

Example 5: Given the decision tree in Fig. 4(b), and bound set $\{a, b\}$, the PARTITION algorithm starts at $c0$, traverses to vertex $d0$, then down to vertex $a0$ (traversing the 0-cofactor of each vertex). Vertex $a0$ represents a bound variable so traverse the 0-cofactor of $a0$: $b0$. The 0-cofactor of $b0$ is a terminal vertex-1, so a 1 is returned. PARTITION moves back to $d0$ and then traverses the 1-cofactor of $d0$ down to the 0-cofactor of $b2$: *don't care* terminal vertex. PARTITION returns "$-$" and moves back to $d0$, creating a new vertex, $d2$ vertex with cofactors $[1, -]$. Calling PARTITION on the 1-cofactor of $c0$ returns a new vertex $d3$ with cofactors $[0, -]$. Because $c0$ is a free variable, a new vertex, $c1$ is created with cofactors $[d2, d3]$. The new diagram that represents the 00 column in the partition matrix is shown in Fig. 4(d).

Because PARTITION and EVAL do not perform any variable swapping, the algorithms are fairly fast. The EVAL and PARTITION algorithms are limited to small bound sets because the number of columns is exponential with respect to the number of variables in the bound set. The restriction of small bound sets is used in many decomposition algorithms, including FPGA synthesis [6] and decomposition for machine learning decomposition [14], [15]. The LPV algorithm does not have the same restriction, but finding the column multiplicity of a incompletely specified function is a NP-hard problem with respect to the number of columns in the partition [16]. To reduce the complexities of finding the column multiplicity of a partition, the bound set size should be restricted, which in essence also restricts the bound set sizes used by the LPV algorithm.

IV. EXPERIMENTAL RESULTS

Tables I and II compare the run-times of the different algorithms presented in this paper on different benchmark sets using a MDD package. The benchmark sets used are the MCNC benchmarks [17] and the multivalued benchmarks [18]. The benchmarks are separated in the tables for readability and to show the difference of timing between each benchmark type. In the tables, the MCNC benchmarks are at the top and the multivalued benchmarks are on the bottom. The MCNC benchmark functions are multiinput and multioutput, completely specified functions. The multivalued benchmarks are all single output, incompletely specified, multivalued input and multivalued output functions.

The units for time are in seconds. Any element in the table that begins with "$>$", states that the program did not complete in the designated time. A "$-$" denotes that the number of bound variables selected was larger than the number of input variables, thus a decomposition could not be done.

Table I compares the EVAL and the LPV algorithms. To make the comparison, we implemented our own multivalued version of the LPV algorithm. The EVAL and LPV algorithms were run using variable bound set sizes of two, four, six, and eight. For each bound set size, the algorithms were run on 100 random partitions and the time to evaluate all 100 partitions is displayed in the table. The set of random partitions was the same for both algorithms. The comparison of these algorithms is the run-time to find 100 partitions, this does not include the time to determine the column multiplicity of each partition. In all cases of two-variable bound sets, the EVAL algorithm is much faster than the LPV algorithm. As the size of the bound sets increases, the LPV algorithm starts to perform better than the EVAL algorithm. This is caused by the number of simultaneous traversals in the EVAL algorithm. When comparing the EVAL and LPV algorithms on the multivalued benchmark functions, the run-times of the LPV algorithm were found to be very high. The reason for the large amount of time is that each of the functional inputs are multivalued which greatly increases the number of possible output combinations for the function. In fact, the LPV algorithm does not complete in the specified time for most of the multivalued benchmark functions, while the EVAL completes for all two- and four-variable bound sets.

Table I shows that the two algorithms are almost equivalent for the binary, completely specified functions, but for multivalued functions the EVAL algorithm performs much better than the LPV algorithm. The algorithmic complexity of the LPV algorithm encoding depends on the number of row output values for a given column, specifically, the complexity is exponential in the number of free variables. Once the LPV algorithm is completed, each encoded integer value must be decoded to determine column compatibilities, which is based on the number of free variables for the decoding and the number of bound variables for checking for column compatibility. The time complexity of the EVAL algorithm is exponential in the number of bound variables. Thus, for small bound sets, the EVAL algorithm should perform well, even for very large multivalued functions. The reason that EVAL did not complete in the specified time with eight-variable bound sets for most of the benchmarks, is that the number of columns in each partition is greater than 20 000.

Table II compares the PARTITION and *cut_level* algorithms. To compare the two algorithms, all two-input bound set partitions of the functions were evaluated. The table shows the initial size (number of vertices in the MDD) before starting the decomposition process. PARTITION and *cut_level* were compared by showing the time to evaluate all two-input bound set partitions and the worst-case size (number of vertices). The two algorithms were also compared on the worst-case number of vertices in the MDD when evaluating a partition. The column size% gives the size percentage of the two algorithms. Table II shows that the PARTITION algorithm is always faster than the *cut_level* algorithm. When comparing the decision diagram sizes created by the two algorithms, the sizes are generally the same, except for the benchmark function *frg1*. We feel that this function fits in the category of reordering that results in a decision diagram of exponential size. While the PARTITION and *cut_level* algorithms usually result in the same size decision diagrams and the same amount of time to complete the task, there are situations where the *cut_level* algorithm results in decision diagrams that are much larger than the PARTITION algorithm.

V. Conclusion

Two new algorithms, EVAL and PARTITION, were created that do not reorder the MDD and, thus, do not cause an exponential increase in size of the MDD by reordering. The advantage of the EVAL algorithm is that the partition table is constructed by rows and determines column compatibilities while constructing the partition table. The LPV algorithm, on the other hand, constructs the partition table and then must determine column compatibilities.

Comparisons were run against the well-known algorithms presented by Lai, Pedram, and Vrudhula. In most cases, the EVAL algorithm is faster than the LPV algorithm, especially on small bound sets. In fact, the EVAL algorithm was able to complete in the specified amount of time on the large multivalued functions while the LPV algorithm could not. EVAL is also much faster than running the PARTITION or *cut_level* algorithms. A restriction of the EVAL and PARTITION algorithms is that they are only practical for partitions with small bound sets. This restriction is actually used in many decomposition strategies, including FPGA decomposition and decomposition for machine learning [14], [15]. Machine learning functions are typically large multivalued, incompletely specified functions where heuristics are used to reduce the search space by only evaluating small bound sets.

References

[1] R. L. Ashenhurst, "The decomposition of switching functions," Bell Laboratories Rep., vol. 16, 1956, pp. III-1–III-72.

[2] ——, "The decomposition of switching functions," in *Proc. Int. Symp. Theory of Switching Functions*, 1959, pp. 74–116.

[3] H. A. Curtis, "Generalized tree circuit-the basic building block of an extended decomposition theory," *ACM*, vol. 10, pp. 562–581, 1963.

[4] ——, *A New Approach to the Design of Switching Circuits*. Princeton, NJ: Van Nostrand, 1962.

[5] R. Brayton, G. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S.-T. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. Ranjan, S. Sarwary, T. Shiple, G. Swamy, and T. Villa, "VIS: A system for verification and synthesis," in *Proc. 8th Int. Conf. Computed-Aided Verification*, R. Alur and T. Henzinger, Eds. New Brunswick, NJ, 1996, pp. 428–432.

[6] Y. T. Lai, K. R. Pan, and M. Pedram, "OBDD-based functional decomposition: Algorithms and implementation," *IEEE Trans. Computer-Aided Design*, vol. 15, pp. 977–990, Aug. 1996.

[7] Y. T. Lai, M. Pedram, and S. B. K. Vrudhula, "BDD-based logic decomposition," Dept. Elect. Eng. Syst., Univ. Southern California, Los Angeles, CA, Tech. Rep., 1992.

[8] ——, "BDD-based decomposition of logic functions with application to FPGA synthesis," in *Proc. Design Automation Conf.*, 1993, pp. 642–647.

[9] R. E. Bryant, "Symbolic Boolean manipulation with ordered binary-decision diagrams," *ACM Computing Surveys*, vol. 24, no. 3, pp. 293–318, 1992.

[10] H. A. Curtis, "Generalized tree circuit," *ACM*, pp. 484–496, 1963.

[11] C. Files, R. Drechsler, and M. Perkowski, "Functional decomposition of MVL functions using a multi-valued decision diagram," in *Proc. Int. Symp. Multi-Valued Logic*, 1997, pp. 27–32.

[12] C. Files and M. Perkowski, "Multi-valued functional decomposition as a machine learning method," in *Proc. Int. Symp. Multi-Valued Logic*, 1998, pp. 173–178.

[13] M. Perkowski, R. Malvi, S. Grygiel, M. Burns, and A. Mishchenko, "Graph coloring algorithms for fast evaluation of Curtis decompositions," in *Proc. Design Automation Conf.*, 1999, pp. 225–230.

[14] C. Files and M. Perkowski, "An error reducing approach to machine learning using multi-valued functional decomposition," in *Proc. Int. Symp. Multi-Valued Logic*, 1998, pp. 167–172.

[15] B. Zupan, M. Bohanec, I. Bratko, and J. Demsar, "Machine learning by function decomposition," in *Proc. ICML-97*, D. H. Fisher, Ed., 1997, pp. 421–429.

[16] M. Perkowski, T. Luba, S. Grygiel, P. Burkey, M. Burns, N. Iliev, M. Kolsteren, R. Lisanke, R. Malvi, Z. Wang, H. Wu, F. Yang, S. Zhou, and J. Zhang, "Unified approach to functional decompositions of switching functions," Portland State Univ., Portland, OR, ser. Tech. Rep., 1995.

[17] MCNC. (1991) Benchmark functions. [Online] Available: ftp://mcnc.mcnc.org/

[18] C. Files. (1999) POrtland Logic Optimization (POLO) group Boolean and multi-valued benchmark functions. [Online] Available: HTTP: http://www.ee.pdx.edu/polo/functions/

On Synchronizable Circuits and Their Synchronizing Sequences

Irith Pomeranz and Sudhakar M. Reddy

***Abstract*—Synchronizing sequences are important in facilitating the test generation process for detectable faults, and in identifying undetectable faults. Synchronizing sequences are also important in determining whether an undetectable fault can be removed from a circuit without affecting its normal operation, i.e., in determining whether a fault is "redundant." In this work, we show a class of faults such that a synchronizing sequence for a faulty circuit can be obtained by repeating the synchronizing sequence of the fault-free circuit. Identification of such faults can be done by simulating the faulty circuits under the repeated synchronizing sequence of the fault-free circuit. We present experimental results to demonstrate the existence of such faults in benchmark circuits.**

***Index Terms*—State diagrams, synchronizing sequences, synchronous sequential circuits.**

I. Introduction

Synchronizing sequences for synchronous sequential circuits are important during test generation for detectable faults and in determining whether a given fault is undetectable [1]–[3]. A sequence that synchro-

Manuscript received April 25, 1998; revised August 5, 1999. This work was supported in part by the National Science Foundation (NSF) under Grants MIP-9357581 and MIP-9725053. This work is based in part on "On Synchronizing Sequences and Test Sequence Partitioning," in *Proc. IEEE VLSI Test Symp.*, Apr. 1998. This paper was recommended by Associate Editor K.-T. Cheng.

I. Pomeranz is with the School of Electrical and Computer Engineering, Purdue University, W. Laffayette, IN 47907-1285 USA.

S. M. Reddy is with the Electrical and Computer Engineering Department, University of Iowa, Iowa City, IA 52242 USA.

Publisher Item Identifier S 0278-0070(00)07478-9.