CUBE DIAGRAM BUNDLES: A NEW REPRESENTATION OF STRONGLY UNSPECIFIED MULTIPLE-VALUED FUNCTIONS AND RELATIONS

S. Grygiel, M. Perkowski, M. Marek-Sadowska †, T. Luba •, L. Jozwiak +,

Portland State University, Dept. of Electrical Engineering, Portland, Oregon 97207,
Tel: 503-725-5411, Fax: 503-725-4882, mperkows@ee.pdx.edu,
† Univ. of Calif., Santa Barbara, Electrical and Computer Engineering Dept.,
Santa Barbara, CA 93106, mms@ece.ucsb.edu,
• Warsaw Univ. of Technology, Dept. of Electronics, Inst. of Telecommunication,
Warszawa, Nowowiejska 15/19, Poland, luba@tele.pw.edu.pl,
+ Faculty of Electronics Engineering, Eindhoven University of Technology,
P.O. Box 513, 5600 MB Eindhoven, The Netherlands, lech@eb.ele.tue.nl

Abstract

Efficient function representation is very important for speed and memory requirements of multiple-valued decomposers. This paper presents a new representation of multiple-valued relations (functions in particular), called *Multiple-Valued Cube Diagram Bundles (MVCDB)*. MVCDBs improve on Rough Partition representation by labeling their blocks with variable values and by representing blocks efficiently. The MVCDB representation is especially efficient for very strongly unspecified multiple-valued input, multiple-valued output functions and relations, typical for Machine Learning applications. [1]

I. Introduction.

Multiple-valued functions and relations that include very many don't cares are becoming increasingly important in several areas of applications such as Machine Learning and Knowledge Discovery [16] and also in combinational and sequential circuit design. It is important to have an efficient representation for such relations. For instance, the successes of many **binary** decomposers depended on appropriate innovative representations of Boolean functions for which some properties could be checked very rapidly: cube calculus [26], spectral transforms [22], decision diagrams [9, 21], and rough partitions [13]. Better representation allows storing larger functions, and also, carrying the appropriate calculations more efficiently. Our work was motivated by multi-valued relation decomposition [18]. The aim was to find such a representation in which the operations of cofactoring, tautology checking, determining intersections, set and all operations necessary for relations had straightforward representations in data structures and could be performed efficiently. With the exception of [7, 8, 13], the representation problem has not been investigated for those **multi-valued** decomposers that have been practically realized in computer programs. A new general-purpose representation of functions and relations proposed here improves on the representations from [7, 8, 13]. It is particularly crucial to speed up the decomposition of large incompletely specified multiple-valued relations and functions. This general-purpose representation takes good properties of MV Cube Calculus (MVCC) [4, 23], Decision Diagrams [1] and Rough Partitions [13, 11, 10, 12, 13], and is related to Boolean and multiple-valued relations [2, 27]. It is especially useful for very weakly specified relations, and allows to efficiently implement algorithms which used MVCC, Decision Diagrams, and Calculus of Rough Partitions in the past.

II. Representation of Incompletely Specified Multi-Valued Functions

Two essentially different representation methods for MV functions are used in programs: Multiple-Valued Cube Calculus (MVCC) [23] and Multiple-Valued Decision Diagrams (MVDD) [14, 5]. These methods have also been extended to incompletely specified functions. Here we will focus on the area that has not been researched until recently: very weakly specified functions, specified by very many variables but with relatively small percentage of care minterms, i.e. input combinations for which function is specified. We call them the *Strongly Unspeci-*

[1] This research was partially supported by the Dutch Technology Foundation (STW) under project EEL 55.3958

cube #	a	b	f	g
0	0,2	1	-	2
1	0,1	0	0,2	0
2	2	0	1,2	0
3	1	1	1,2	2

Table 1: *Table of MV relation. Rows correspond to MV cubes. Output variable g is a function.*

fied Functions. Multiple-valued functions of this type occur in Machine Learning (ML) [16], Knowledge Discovery in Databases (KDD) [6], and to a lesser degree in Finite State Machine design (for instance, the 1993 FSM MCNC benchmark *s8* has 85% next state don't cares and *kirkman* has 71% output don't cares).

Below, we will give an overview of incompletely specified function representations considered previously in literature, and explain why these representations are inadequate for our class of problems. Observe, that a discrete function, or a *discrete relation*, can be represented as a two-dimensional table (tables like this are not a part of the representations, and we use them solely for ease of explanation). An example is shown in Table 1. There, the columns correspond to the input variables (a, b), a 3-valued output relation f, and the 3-valued output function g. The input variables describe the domain, and the output variable(s) describe the co-domain. A row stores an element of the domain and a set of the corresponding function values. Rows are enumerated, each of them is assigned a "row number". For a *function*, each minterm has a single value in the co-domain, for a *relation*, minterm may correspond to many values in the co-domain. For instance, the row number 3 in Table 1 states that minterm 11 maps to values 1 or 2 in the 3-valued output relation f ($f(11) = 1$ **or** $f(0100) = 2$). One can select any of these values during minimization, whichever simplifies the final description better. Such entry in the table is called *generalized don't care.* For function g, each position in the column has a *single* value. Observe that out of 6 minterms of 2 variables the table includes only 4 care cubes (minterms in this case) - the remaining 2 don't care minterms are not listed. This is an implicit way of representing standard output don't cares of all output variables. When output don't care occurs only for some outputs, an explicit don't care symbol "-" is used for them. For instance, for variable f in row 0. Symbol "-" means all possible values of this variable, this is a "standard don't care", representing "totally unknown" values in ML applications.

Let us denote by n the number of input variables and by m the number of output variables. The next generalization of a table representation is to allow for input don't cares. This means, the entries in input variables can take many values. In particular, if all values of a given variable are taken, a standard symbol "-" is used. In such a case, rows correspond to *MV cubes*, i.e., certain groups of MV minterms form complete k-dimensional sub spaces of the n-dimensional hypercube, $k \leq n$. When table has many outputs $(m > 1)$ it stores *multi-output MV care cubes.* A *care cube* is one that has at least one output variable which is not a standard don't care "-". (It is possible to generalize this representation further by adding more columns corresponding to *intermediate (auxiliary) variables.*

The most general relation that can be described using the table representation, has MV input variables with different number of values each and several MV outputs with different number of values each. It may also have sets of values for positions in inputs columns, and sets of values in positions for output variables (meaning a separate relation for each output with the inputs). One can argue that such tables are already realized in MVCC: the rows (without their numbers) would correspond to cubes in an array of cubes. The disadvantage of cube calculus is, however, that large initial multilevel netlists or BDDs may produce too many cubes after flattening, so that their cube arrays cannot be stored. Even for initial data in forms of large arrays of cubes in ML or controller design applications, the cubes can be too slow to manipulate and alternative representation may improve considerably the processing speed.

In our approach we will create MVCDB representation from cubes, in such a way, that during and after its creation, the MVCDB may occupy less storage than would be occupied by the corresponding set of cubes. This is because MVCDBs can be created incrementally, i.e. by reading a row at a time and updating the data structure.

Below, we will first review the method to store MV functions, which will conceptually refer to the table presented above, but which stores the data vertically, not horizontally. This method represents functions with Rough Partitions. Next, we will improve on this representation and generalize to relations. An early attempt to improve on the MVCC representation is presented in [13]. MV functions are represented there by *Rough Partitions* (r-partitions, or RP) for their variables. There are two blocks for a binary variable, and K blocks for a K-valued variable. An RP stores the table columnwise. To every K-valued variable corresponds a column of the table. In RP with minterms as rows, every K-valued variable induces with its values a partition on the set of rows to blocks. For instance, in Table 1 the primary variable b is the header of column 1001. Partition for primary input variable b is: $\Pi(b) = \{B_0,\ B_1\} = \{\overline{0,3},\ \overline{1,2}\}$, rough-partition for output variable f is: $\Pi(f) = \{B_0, B_1, B_2\} = \{\overline{0,1},\ \overline{0,2,3}, \overline{0,1,2,3}$.

The blocks may overlap. For instance an input binary don't care is treated as 0 and 1, so the number of a row with a "-" for a binary variable a is included in both blocks of the partition for this variable. The RP representation is a set of rough partitions for all variables (input and output) of the function. A similar, double-feature representation of lists of lists and lists of characteristic functions (bit vectors) was used in [7, 8] for sequential machines.

Observe, that although the RP is a really new and interesting idea, it has the following major drawbacks: **(1)** RP does not represent a function, but only certain abstraction of a function. This is because, for blocks in a partition of variable X, the values of the variable are not stored together with the blocks. Thus, some information of a function is lost when it is represented by a set of partitions for its variables. **(2)** Similarly to arrays of cubes, the RP implementation, as described in [13], is a flat list representation, with all known disadvantages of such representations. **(3)** RP do not represent relations. We believe that for these reasons the "pure" partition-based representations of functions and state machines [13, 7, 8] have been limited in the past in their applications and popularity.

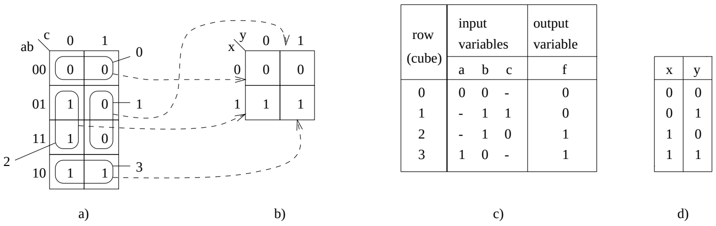
In both binary and MV cases, an efficient representation for strongly unspecified relations (and even functions) has not been yet proposed and this paper tries to fill this gap. We call our new representation the *MV Cube Diagram Bundles (MVCDB)*: *Cube* - because it operates on cubes as atomic representations; *Diagrams* - because it uses Decision Diagrams (of any kind) to represent sets; *Bundles* - because several diagrams and data are bundled together to specify multi-output *Multiple-Valued Relation.* The MVCDB representation is general and can be applied to both binary and multiple-valued functions (Finite State Machines) and relations (non-deterministic Finite State Machines) in the same way.

III. Multiple-Valued Cube Diagram Bundles to Represent Functions and Relations

A multi-output MV cube is represented as a row of the conceptual table introduced in the previous section. Below, rows and cubes will be treated interchangeably. Originally, the cubes are expressed in terms of the *primary input variables.* In MVCDB representation, each cube is encoded with new binary variables called the *secondary variables.* To each cube corresponds a minterm in secondary variables. If not specified otherwise, the binary codes of the secondary variables correspond to the decimal numbers enumerating the rows in the table.

Example 1. A Kmap with primary input variables a, b, and c is shown in Fig. 1a. As the result of encoding of primary cubes with secondary input variables, x and y, a new map, Fig. 1b, is created. Encoding of variables x and y corresponds in this case to natural binary codes of the numbers of rows. Figures 1a and 1b show how cubes of the first map are encoded into minterms of the secondary map. In general, this encoding can be arbitrary, and can be of a non-minimum length. The table for the function from Fig. 1a is shown in Fig. 1c, and the encodings of its rows to secondary input variables is shown in Fig. 1d.

For each primary input variable VAR a set of Boolean functions is created; there are as many functions as the values that the variable can take. These functions are



row (cube)	input variables a b c	output variable f
0	0 0 -	0
1	- 1 1	0
2	- 1 0	1
3	1 0 -	1

x	y
0	0
0	1
1	0
1	1

Figure 1: *Mapping from primary to secondary variables for Example 1: (a) original function with primary inputs, (b) secondary space with secondary input variables x and y, (c) table of function f, (d) encoding of primary cubes.*

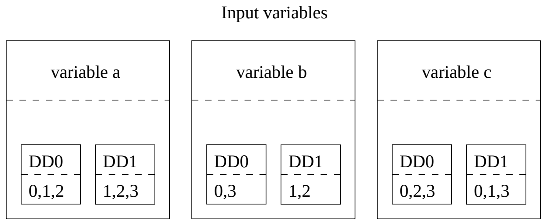


Figure 2: *Input part of a data structure for the MVCDB from Example 1*

represented as: $DD_{VAL}(VAR) \mid VAL = 0, 1, ..K_{VAR}$, where K_{VAR} is the number of values of variable VAR. The ON-set of each $DD_{VAL}(VAR)$ represents a set of minterms on secondary variables that have value VAL for variable VAR. The number of rows of the table determines the number of minterms in secondary variables and thus the size of the corresponding set called Value DD. For instance, in the MVCDB from Fig. 2, which corresponds to the function from Example 1, by "DD for set {0,1,2}" we understand a Value DD that represents the set of natural numbers {0,1,2} encoded in (binary) secondary input variables as {00,01,10}, respectively. Assuming secondary variables x and y, this yields $\{\overline{x}\,\overline{y}, \overline{x}y, x\overline{y}\}$. All the Value DDs are built together from columns of the conceptual table, scanned row after row. They are shared and ordered.

For consistency with MV logic, the ON cube will be called a $\mathbf{VAL}_1$-cube, and the OFF cube, a $\mathbf{VAL}_0$-cube. In a single binary-output function F, each **true minterm** of the function represented by $DD_1(F)$ of MVCDB(F) corresponds to the table's row, i.e., to an ON cube in function F on *primary variables.* Similarly, each **false minterm** in $DD_0(F)$ corresponds to an OFF cube in function F on primary variables. In the case of a MVCDB representing a K-valued logic function, each s-valued minterm of function represented by $DD_s(F)$ of the MVCDB is a VAL_s-cube, s=0,...$K-1$, on primary variables.

MVCDB representation (see Fig. 2) of function F may be seen as a hierarchy of labeled sets. At the highest level, there are sets labeled by function's input and out-

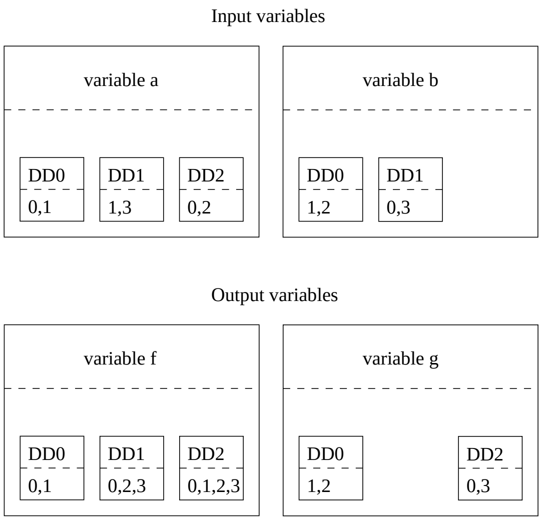


Figure 3: *Data structure for a MVCDB from Table 1 to Example 2*

put variables (variable a, variable b, variable c, on Fig. 2). Each of them consists of sets labeled by the variable's value (DD0 corresponds to value 0, DD1 corresponds to value 1). Each of sets labeled by the variable's value contains corresponding row numbers and is represented by a decision diagram.

Example 2. An MVCDB for function from Table 1 is shown in Fig. 3. Here $DD_0(a) = DD_0(f)$, $DD_0(b) = DD_0(g)$, and $DD_1(b) = DD_2(g)$ so these sets don't need to be repeated in the function F representation saving memory space. Standard don't cares for f are treated as a generalized don't care and all their values are stored in the Value DDs. Observe also the pointer to $\emptyset$ for the non-used value 1 of output g.

Now we formally define the MVCDB.

Definition. Given is MV relation R in a form of a table. *The MVCDB representation* of this table is the set of rough partitions on rows for all its input and output variables, in which all blocks are labeled with respective values of these variables. Thus, MVCDB is a set of labeled sets. For efficiency of processing and storage, this set is represented in a compressed form. Any method to represent sets is possible. In particular, these sets can be represented as BSs, BDDs, BMDDs, EVDDs, KFDDs, K*BMDs, ZBDDs, etc. [15].

IV. MVCDBs allow efficient manipulation and storage of functions and relations

If an (implicit) cube has standard output don't cares for **all** its outputs, it is not stored (explicitly) in the MVCDB at all. This means that only care cubes are stored. Don't care minterms are represented implicitly, because everything that is not a care is implied to be a don't care. This means, for large functions and relations with many don't cares, a big saving of both storage and processing time, when compared to the representations that store don't cares explicitly (such as MVCC in Espresso-MV). Also, a MVDD has to store pointers to the terminal node "DC". If there are L disjoint DC cubes in a map, there would be L such pointers, and this number can be exponential in the number of input variables. Moreover, MVDD requires a good ordering of MV input variables, which has not been successfully solved and can lead to prohibitively large diagrams. In contrast, the size of MVCDB is in the worst case of the order of the number of cares, so it does not depend on the location of the don't cares. In addition, for MVCDB, the encoding with secondary variables is used to decrease the sizes of the DDs. If the secondary variables are binary, the efficient binary BDD packages based on sifting or other variable ordering techniques can be used. In case of using MVCDBs to represent relations, the generalized don't care positions are stored in an efficient way, because they are treated in the same way as the input don't cares, and the sharing of subsets is used between all the variables. Also the input and output variables are represented uniformly.

It is well-known, that there are functions, such as parity, for which BDDs are obviously better, and there are other functions, such as the one shown by Devadas [3] (or that occur in ML, logic or controller design [25]), that are more efficiently described using an array of cubes. It can be shown that with good selection of Value DDs encoding, in these two extreme worst cases the MVCDBs are comparable in size to the *better* representation of the two: arrays of cubes, or BDDs. **(1)** One extreme example is a completely specified binary function, similar to parity, and with many input variables. Obviously, in this case, a BDD is better than an array of cubes, because the BDD has the polynomial number of nodes, and the array of cubes has an exponential number of cubes. In this case the original variables are selected as the secondary variables for the MVCDB. Thus the size of the Value DD for the ON-set of the output variable is the same as that of the BDD for this function. All the Input Value DDs have one node each. So, disregarding a small overhead in the top lists of the MVCDB data structure, both representations are comparable in size. **(2)** For the other extreme case, let us consider a binary function like those discussed in [3] that have polynomial number of cubes and exponential number of nodes in BDD. When the function is specified by cubes, it has n variables and k cubes, $k << 2^n$. Very conservatively estimating: in the worst case there are $2(n+1)$ Value DDs, each with k nodes. So, the total number of DD nodes is $O(2nk)$ while the number of nodes in the BDD would be $O(2^n)$. Examples of multi-output MV relations can be constructed for which the advantage over MVDDs would be dramatic for large values of n and k. It seems, that there exist practical functions with similar, although not that extreme properties [25]. To this category belong functions with many cubes and many variables, but with still very small ratio of cares to don't cares. This category includes the same kind of functions as those from the ML benchmarks, but with even larger k, n and number of terms than in all the functions from U.C. Irvine benchmarks. We expect that for larger multi-valued functions or relations the advantages of MVCDBs will be even more clearly observable.

We implemented a decomposer, GUD-MV, for MV relations, [18] which splits up a relation into smaller blocks in order to minimize the overall relation complexity. Complexity of MV relation is determined by a parameter called **Relation Cardinality** (RC) and defined as follows:

Definition. **Relation Cardinality** (RC) for MV relation with a set of inputs $X = \{x_0, x_1, \ldots, x_n\}$ and set of outputs $Y = \{y_0, y_1, \ldots, y_m\}$ is defined by the following formula:

$$RC = \Big(\prod_{x_i \in X} m_{x_i} \Big) \sum_{y_j \in Y} \log_2 m_{y_j}$$

where: m_{x_i} is multiplicity of variable $x_i \in X$,
m_{y_j} is multiplicity of variable $y_j \in Y$.

The above definition extends **Decomposed Function Cardinality** (DFC) [20] on MV relations and is directly related to the amount of information the relation could possibly handle.

Concise and elegant solutions were obtained and verified (for instance, for the difficult example *trains*). Some benchmarks are in Tables 3 and 4. Benchmarks run on SPARC-10 workstations. Times listed are user times. Result of comparison for selected MCNC benchmarks is shown in Table 4. [1]-Results not available for this benchmark, [2]-Program limited to 32 input and 32 output variables, TRADE - Program designed at PSU [26], MISII - Program designed at UC Berkeley, DSGN174 - Program designed by B. Steinbach [24], GUD-MV - Program designed at PSU. As we can see from the table, complexity of the functions obtained using GUD-MV is in most cases smaller than for other decomposers. And smaller complexity translates directly to smaller circuit area and power consumption in circuit design and to better generalization properties in ML. Table 3 shows the number of BDD nodes for few ML benchmarks. Note that variables are multi-valued, so even benchmarks *breastc* and *balance* are strongly unspecified. Observe that even for the medium size real-life data such as *breastc* the percent of don't cares is very high (for nine 10-valued variables there were $10^9 - 699$ don't cares and only 699 cares. Thanks to our MVCDB representation GUD-MV is fast. For instance, the benchmarks *zoo, shuttle, lenses* and *trains* took with BDDs only 9.9, 1.8, 0.3 and 1.8 seconds, respectively. Other benchmarks, such as *breastc* and *balance* are slower (1880.0s and 44.3s) but these are quite large examples and the decomposer looks to many partition candidates.

All MV operations on MVCDBs use *set-theoretical operations* on the corresponding sets representing blocks. Therefore, any computer package for representing and manipulating sets (and in particular any DD package that allows set-theoretical operations), can be used to implement MVCDBs with no modification: for instance the packages for BMDDs, EVDDs, KFDDs, K*BMDs, ZBDDs, etc. [15]. We plan to compare the efficiency and storage requirements for MVCDBs with various data structures for blocks. Especially, we plan to experiment with these new classes in our C++ based MVCDB package.

V. Conclusions

In this paper we report the following observations related to application of multiple-valued logic synthesis to Machine Learning and KDD.

1. The data occuring in these applications are *distinctly different* from the most data taken from circuit design. Some of these data are relations and all are strongly unspecified.

2. These data should be represented in a special way in a computer. The improved Rough Partition representation introduced here, allows not only for faster Ashenhurst/Curtis decomposition [18] but also for many other types of decomposition, such as for instance the binary AND/OR/EXOR decompositions to two-input gates [24]. Also, our new MVCDB representation does not lose any information, so it is a representation of the function, and not only some abstraction of the function (as is the RP representation). The MVCDB representation is especially good when there exist some variables with few, and other variables with many values, and relations are weakly specified, a situation that is typical in ML. Relations allow to elegantly formalize many aspects of generalized decompositions, and MVCDBs allow to directly translate algorithms to data structures.

3. Finally, for the case of large initial data, the paper proposed to represent the blocks in a compressed way in order to further save on the memory and gain on the processing speed. We compared two data structures for the compressed blocks: GNU++ Bit Sets and Univ. of California Berkeley standard BDDs. For some benchmarks, GUD-MV can decompose functions too large to be solved by competing methods. The results of comparison, as well as more detailed presentation of operations on MVCDBs and benchmarks are presented in [19]. Although large U.C. Irvine benchmarks seem still to favour Bit Sets, we believe DDs will be better for even larger benchmarks. The concept of MVCDB does not specify how the sets should be represented, so any new improved representation can be used in the future.

References

[1] R.E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *Trans. on Comput.*, Vol. C-35, No. 8, pp. 667-691, 1986.

[2] R. Brayton and F. Somenzi, "An Exact Minimizer for Boolean Relations," Proc. of ICCAD, pp. 316-320, 1989.

[3] S. Devadas, "Comparing Two-Level and Ordered Binary Decision Diagram Representations of Logic Functions," *IEEE Trans. on CAD*, Vol. 12, No. 5, May 1993, pp. 722-723.

[4] D.L. Dietmeyer, "Logic Design of Digital Systems," *Allyn and Bacon, Boston, MA,* 1971.

[5] R. Drechsler, "Verification of Multi-Valued Logic Networks," *Proc. 26th ISMVL'96*, May 29-31, Santiago de Compostela, Spain, 1996, pp. 10-15.

[6] J. Han, "Data Mining Techniques", *Proc. 1996 ACM-SIGMOD Int'l Conf. on Management of Data (SIGMOD'96)*, Montreal, Canada, June 1996 (Tutorial).

[7] L. Jozwiak, and F. Vankan, "Bit Full Decompositions of Sequential Machines: Algorithms and Results," *Can. Conf. Electr. Comp. Engn.,* Sept. 17-20, 1989, pp. 1010-1014, Montreal, CA.

[8] L. Jozwiak, "Simultaneous Decompositions of Sequential Machines," *Microprocessing and Microprogramming,* Vol. 30, pp. 305-312, 1990.

[9] Y.T. Lai, K.R. Pan, M. Pedram, and S. Vrudhula, "FGMap: A Technology Mapping Algorithm for Look-up Table Type FPGA Synthesis," *Proc. 30-th DAC,* pp. 642-647, 1993.

[10] T. Luba, and J. Rybnik, "Algorithmic Approach to Discernibility Function with Respect to Attributes and Objects Reduction," *Found. of Comp. and Dec. Sciences,* Vol. 18, No. 3-4, pp. 241-258, 1993.

[11] T. Luba, M. Mochocki, and J. Rybnik, "Decomposition of Information Systems Using Decision Tables," *Bull. Polish Acad. Sci., Techn. Sciences,* Vol. 41, No.3, 1993.

[12] T. Luba, R. Lasocki, and J. Rybnik, "An Implementation of Decomposition Algorithm and its Application in Information Systems Analysis and Logic Synthesis," *Intern. Workshop on Rough Sets and Knowledge Discovery,* pp. 487-498, Banff 1993,

[13] T. Luba, "Decomposition of Multiple-Valued Functions," *Proc. 25th ISMVL*, pp. 256-261, 1995.

[14] D.M. Miller, "Multiple-valued logic design tools," in *Proc. ISMVL,* pp. 2-11, 1993.

[15] S. Minato, "Graph-Based Representations of Discrete Functions," *Proc. Reed-Muller'95 Workshop*, Chiba, Japan, August 1995, pp. 1-10.

[16] M. Perkowski, T. Ross, D. Gadd, J. A. Goldman, and N. Song, "Application of ESOP Minimization in Machine Learning and Knowledge Discovery," ibidem, pp. 102-109.

[17] M. Perkowski, "A New Representation of Strongly Unspecified Switching Functions and its Application to Multi-Level AND/OR/EXOR Synthesis," ibidem, pp. 143-151.

[18] M. Perkowski, M. Marek-Sadowska, L. Jozwiak, T. Luba, S. Grygiel, M. Nowicka, R. Malvi, Z. Wang, and J. Zhang, "Decomposition of Multiple-Valued Relations," *Proc. ISMVL'97.*

[19] S. Grygiel, M. Perkowski, M. Marek-Sadowska, L. Jozwiak, T. Luba, "full version of this paper".

[20] T. D. Ross, M.J. Noviskey, T.N. Taylor, D.A. Gadd, "Pattern Theory: An Engineering Paradigm for Algorithm Design," *Final Technical Report WL-TR-91-1060,* Wright Laboratories, USAF, WL/AART/WPAFB, OH 45433-6543, August 1991.

[21] T. Sasao, "FPGA Design by Generalized Functional Decomposition," *in "Logic Synthesis and Optimization,"* T. Sasao. (Ed), Kluwer Academic Publishers, pp. 233-258, 1993.

[22] V.Y. Shen, A. C. McKellar, and P. Weiner, "An Fast Algorithm for the Disjunctive Decomposition of Switching Functions," *IEEE Trans. on Comput.*, Vol. C-20, No. 3, pp. 304-309, March 1971.

[23] N. Song, and M. Perkowski, "Minimization of Exclusive Sum of Product Expressions for Multi-Output Multiple-Valued Input Switching Functions," *IEEE Trans. on CAD,* Vol. 15, No. 4, pp. 385 - 395, April 1996.

[24] B. Steinbach, and A. Wereszczynski, "Synthesis of Multi-Level Circuits Using EXOR-Gates," *Proc. Reed-Muller'95 Workshop,* Chiba, Japan, August 1995, pp. 161-168.

[25] Papers of Steinbach/Hesse, Kempe, Rohde/Barthel, and discussions at the *2nd Workshop Boolesche Probleme*, 19-20 September, Freiberg, Sachsen, 1996.

[26] W. Wan, and M. Perkowski, "A New Approach to the Decomposition of Incompletely Specified Multi-Output Function Based on Graph Coloring and Local Transformations and Its Application to FPGA Mapping," *Proc. Euro-DAC*, pp. 230 - 235, 1992.

[27] Y. Watanabe, and R.K. Brayton, "Heuristic Minimization of Multiple-Valued Relations," *IEEE Trans. on CAD.*, Vol. 12, No. 10, pp. 1458-1472, October 1993.

	input file		BDD set representation		
	# of in/out	# of cubes	# of BDDs	# of nodes	# nodes per BDD
zoo	16/1	101	43	1080	25.1
shuttle	6/1	15	18	80	4.4
breastc	9/1	699	92	5300	57.6
balance	4/1	625	23	768	33.4
lenses	4/1	24	10	40	4.0
trains	32/1	10	107	360	3.4

Table 2: *BDD set representation, number of BDD nodes for MV functions from ML Benchmarks.*

		TRADE		MISII		DSGN174		GUD-MV	
	in/ out	RC	time [s]	RC	time [s]	RC	time [s]	RC	time [s]
5xp1	7/10	496	2.3	384	4.2	292	2.7	227.7	10.9
9sym	9/1	640	29.9	984	17.8	400	8.7	102.2	26.2
b12	15/9	412	3.6	[1]	[1]	[1]	[1]	196.9	11.2
bw	5/28	1148	9.5	[1]	[1]	[1]	[1]	577.3	21.1
clip	9/5	[1]	[1]	[1]	[1]	[1]	[1]	466.9	57.6
con1	7/2	80	0.8	68	0.3	60	0.4	70.0	2.2
ex5p	8/63	[2]	[2]	3720	18.1	1560	4126.8	1956.0	207.6
inc	7/9	[1]	[1]	[1]	[1]	[1]	[1]	369.4	13.9
misex1	8/7	472	4.1	208	1.4	224	1.9	229.0	8.7
rd53	5/3	120	1.2	96	1.3	84	0.5	59.9	1.8
rd73	7/3	320	2.5	352	14.1	256	3.5	113.0	13.1
rd84	8/4	508	32.6	672	119.7	320	7.3	170.9	32.5
sao2	10/4	1848	31.5	516	9.6	468	13.9	440.6	46.8
sqrt8	8/4	[1]	[1]	[1]	[1]	[1]	[1]	136.0	7.2
squar5	5/8	228	2.1	[1]	[1]	[1]	[1]	135.9	3.6
xor5	5/1	16	0.1	[1]	[1]	[1]	[1]	16.0	0.4

Table 3: *Results on binary function MCNC benchmarks*