New Compact Representation of Multiple-Valued Functions and Relations

Stanislaw Grygiel, Marek Perkowski
Portland State University
Department of Electrical Engineering
Portland, OR 97207

Abstract

In this paper we present a new data structure for representing Multiple-valued relations (functions in particular) both completely and incompletely specified. Relations are represented by Labeled Rough Partitions, structure similar to Rough Partitions introduced in [9] but extended with labels to store the full information about relations. We present experimental results from comparison of our data structure to Binary Decision Diagrams (BDDs) on binary functions (MCNC benchmarks) showing its superiority in terms of memory requirements in 73% cases. The new representation can be used to a very large class of multiple-valued, completely and incompletely specified functions and relations, typical for Machine Learning (ML) and complex FSM controller optimization applications.

1. Introduction

Until very recently, experiences gained in the area of logic synthesis were used only to minimize and optimize the hardware of digital circuits. In the last few years an increased trend occurred to apply these methods also in image processing, machine learning, knowledge discovery, data-base optimization, AI, image coding, automatic theorem proving and verification. Multiple-valued (MV) relations (functions in particular) which include very many don't cares are becoming increasingly important, especially in these new areas of application [11]. It is very important to have a good representation for such relations. By good representation we understand one that is compact and allows fast processing. For instance, success of many **binary** decomposers depends on appropriate innovative representation of Boolean functions: cube calculus [19], spectral transforms [14], decision diagrams [8, 13], or rough partitions [9]. Better representation allows storing larger functions, and also, to carry efficiently appropriate calculations.

Three essentially different representation methods for MV functions have been successfully used in logic synthesis programs and AI applications:

Multiple-Valued Cubes (MVC, using positional notation) [18], store cubes of the table one by one (row by row), value by value (linearly) and apply *cube calculus* for cube operations.

Multiple-Valued Decision Diagrams (MVDD) [12],[16],[4],[7] store cubes in a Directed Acyclic Graph (DAG). Each DAG level corresponds to a different variable, the number of node's children is equal to the number of values a variable can take.

Rough Partitions (r-partitions), this representation [9] stores the table column-wise, and not row-wise as MVC does. In r-partition every variable (a column of a table) induces a partition of the set of rows (cubes) to blocks, one block for each value the variable can take (there are two blocks for a binary variable, and k blocks for a k-valued variable).

Cube representation seems to be superior in problems with a limited number of levels, such as sum-of-products (SOP) or exclusive-sum-of-products (ESOP) synthesis. The disadvantage of cube representation is that large multilevel netlists or BDDs may produce too many cubes after flattening, so that their cube arrays can not be stored. Even if the initial data is in form of large arrays of cubes in ML or controller design applications cubes may be too slow for effective manipulation and alternative representations may considerably improve the processing speed.

Decision Diagram representation seems to be superior for general-purpose Boolean function manipulation, simulation, tautology, technology mapping, and verification, but can be exponential for functions of certain classes. For some classes of functions such as parity, decision diagram storage requirement is polynomial in terms of the number of variables. For other functions however, as shown in [5] or that occur in ML, logic or controller design [17], it is exponential.

Rough Partition representation is an interesting and novel idea but it doesn't really form a representation of a function. Since the values of a variable are not stored together with partition blocks, the essential information on the

function is lost and the original data can not be recovered from it. Also, the rough partition implementation as described in [9], is a flat list representation with its all known disadvantages.

None of the above representations addresses the problem of binary or MV **strongly unspecified** functions and relations which occur in Machine Learning (ML) [11], Knowledge Discovery in Databases (KDD), Artificial Intelligence (AI), and Finite State Machine (FSM) and controller design. Although the logic and FSM methodologies that produce a very high percent of don't cares are not very popular yet, there exist practical industrial applications with more than 95% of don't cares [15]. In contrast, benchmarks with more than 99% of don't cares are common in ML.

With the exception of [9] and [6], the representation problem has not been addressed for **MV decomposers** and other synthesis programs. The data structure presented in this paper is particularly useful for the decomposition of incompletely specified MV **functions and relations**.

The paper is organized as follows. Section 2 defines *generalized values* which allow for representation of MV cubes and relations. Section 3 defines *MV cubes*. Our representation is built from a relation given in the form of MV cubes. MV cubes are also used to label partition blocks of the representation. Sections 4 and 5 introduce *Labeled Rough Partition* (lr-partition) representation. Section 6 discusses memory requirements for two different representations of partition blocks: BDDs and Bit Sets (BS). Section 7 presents results of testing and Section 8 concludes the paper.

2. Generalized values

The following notation will be used in the paper:

$X = \{x_i\}$	set of MV input variables.
$Y = \{y_i\}$	set of MV output variables.
$\|X\|$	cardinality of set X.
$\|x\|$	cardinality of variable $x \in X$.
$Q_x = \{q_{ix}\}$, $i = 1, \cdots, \|x\|$	set of symbols for $\|x\|$-valued variable x.

Definition 1 The **generalized value** V of variable x is defined as a subset of set Q_x, $V \subseteq Q_x$, and denoted by $V(x)$. □

$V(x)$ can be any subset of Q_x and the number of possible values $V(x)$ can take for a given Q_x is equal to $2^{|x|}$ ($|x| = |Q_x|$). The set of possible values $V(x)$ can take is a power set of of the set Q_x.

The above definition extends the notion of MV variable value and defines it as a set of symbols (classical, one symbol value is a special case of that definition). In particular $V(x) = Q_x$, which is MV equivalent of binary don't care (variable x can be assigned any symbol from Q_x), and $V(x) = \emptyset$ (none of the symbols can be assigned to the variable).

Definition 1 allows also for expressing situations where there is an uncertainty on what value to assign to a variable but the set of acceptable values (symbols) can be clearly specified. For instance: "the color was red or yellow but not black or white". An example of application in logic synthesis area is a modulo-3 counter that counts in sequence $s0 \to s1 \to s2 \to s0$ and if the state $s3$ happens to be the initial state of the counter, counter should transit to any of the states $s0, s1, s2$, but not to the state $s3$ itself.

It also allows for representing MV relations in Machine Learning. For instance: for a given set of attribute values, cell has been classified as cancerous by one expert and as non-cancerous by another and both experts opinions need to be taken into account in the data analysis. Examples of MV relations are benchmarks `hayes`, `flare1`, `flare2` from U.C. Irvine Machine Learning repository.

Generalized values for input variables are already known from cube calculus but generalized values for output variables are a new concept which allows for representation and manipulation of relations.

3. Multiple-valued Cubes

Definitions in this section are based on cube calculus definitions but extend them on the case of cubes based on different sets of variables.

Definition 2 (MV cube) Let $V_{x_k} = \{V_i(x_k)\}$ be a set of all possible MV values of variable x_k. MV cube based on the set of variables X is defined as an element of the Cartesian product $V_{x_1} \times V_{x_2} \times \ldots \times V_{x_n}$. □

In other words MV cube is a vector of generalized values $\{V(x_i)\}$.

Later in this paper we will denote a cube by c and a set of cubes by C. We will use notation $c(X), C(X)$ to specify a cube and a set of cubes based on the set of variables X. If $x \notin X$ we will assume that $V(x) = \emptyset$ in cube $c(X)$.

$V(x) = \emptyset$ can be used for representing situations where variable x is not present in a given cube. An example from ML domain is the well known Michalski's train benchmark which describes a set of 10 trains. A set of attributes corresponds to every car in the train. Since the number of cars vary from train to train some cubes (trains) contain attributes which correspond to non existing cars and can be assigned value $V(x) = \emptyset$. Another application are output variables of multi-output functions and relations corresponding to unspecified cubes ('~' in Espresso format).

Definition 3 (proper and improper cubes) Cube $c(X)$ will be called improper if there exists $x \in X$ such that $V(x) = \emptyset$. Otherwise cube is called proper. □

Example 1 improper cube: $\{\{1,3\},\{0,1\},\emptyset,\{1\}\}$ proper cube: $\{\{1,3\},\{0,1\},\{1\},\{1\}\}$ □

Definition 4 (minterm) Cube $c(X)$ is called minterm if for every $x \in X, |V(x)| = 1$. □

Example 2 minterm: $\{\{3\},\{0\},\{1\},\{1\}\}$ □

Definition 5 (cube containment) It is said that cube $c_1(X_1)$ is contained in cube $c_2(X_2)$, $c_1(X_1) \subseteq c_2(X_2)$, if for every $x \in X_1 \cup X_2$, $V_1(x) \subseteq V_2(x)$ and $V_1(x), V_2(x)$ are values of the variable x in cubes c_1 and c_2 respectively. □

Example 3 Let $X_1 = \{x_1, x_2, x_3, x_4\}$ and $X_2 = \{x_1, x_2\}$. Then cubes $c_1(X_1) = \{\{3\},\{0\},\{1\},\{1\}\}$ and $c_1(X_2) = \{\{1\},\{0\}\}$ are contained in cube $c_2(X_1) = \{\{1,3\},\{0,1\},\{1\},\{1\}\}$, but $c_1(X_2)$ is not contained in $c_1(X_1)$, $c(X_2) \not\subseteq c_1(X_1)$. □

Definition 6 (cube intersection) The intersection of cubes $c_1(X_1)$ and $c_2(X_2)$ is the cube $c_3(X_3) = c_1(X_1) \sqcap c_2(X_2)$, $X_3 = X_1 \cup X_2$, such that for every $x \in X_3$, $V_3(x) = V_1(x) \cap V_2(x)$, and $V_1(x), V_2(x), V_3(x)$ are values of the variable x in cubes c_1, c_2 and c_3 respectively.

Example 4 Let $X_1 = \{x_1, x_2, x_3, x_4\}$, $X_2 = \{x_1, x_2\}$, $c_1(X_1) = \{\{3\},\{0\},\{1\},\{1\}\}$, $c_2(X_1) = \{\{1,3\},\{0,1\},\{1\},\{1\}\}$, and $c_1(X_2) = \{\{1\},\{0\}\}$. Then $c_2(X_1) \sqcap c_1(X_2) = c_3(X_1) = \{\{1\},\{0\},\emptyset,\emptyset\} = \{\{1\},\{0\}\} = c_1(X_2)$, and $c_1(X_1) \sqcap c_2(X_1) = c_1(X_1)$. □

Definition 7 (supercube) The supercube of cubes $c_1(X_1)$ and $c_2(X_2)$ is the cube $c_3(X_3) = c_1(X_1)\$c_2(X_2)$, $X_3 = X_1 \cup X_2$, such that for every $x \in X_3$, $V_3(x) = V_1(x) \cup V_2(x)$, and $V_1(x), V_2(x), V_3(x)$ are values of the variable x in cubes c_1, c_2 and c_3 respectively. □

Example 5 Let us take the cubes defined as in Example 4. Then $c_2(X_1)\$c_1(X_2) = c_2(X_1)$ and $c_1(X_1)\$c_2(X_1) = c_2(X_1)$. □

Definition 8 (cube merging) Let $c_3(X_3) = c_1(X_1)\ \$\ c_2(X_2)$. We can say that cube $c_3(X_3)$ is a merge of cubes $c_1(X_1)$ and $c_2(X_2)$, $c_3(X_3) = c_1(X_1)\$\$c_2(X_2)$, iff there exists only one $x \in X_3$ such that $V_1(x) \neq V_2(x)$. □

Example 6 Let $X_1 = \{x_1, x_2, x_3, x_4\}$, $X_2 = \{x_1, x_2, x_3\}$, $c_1(X_1) = \{\{3\},\{0\},\{1\},\{1\}\}$, $c_2(X_1) = \{\{1\},\{0\},\{1\},\{1\}\}$, $c_1(X_2) = \{\{1\},\{0\},\{1\}\}$, and $c_2(X_2) = \{\{3\},\{0\},\{1\}\}$. Then $c_1(X_2)\$\$c_2(X_1) = \{\{1\},\{0\},\{1\},\{1\}\} = c_2(X_1)$ and $c_2(X_1)\$\$c_1(X_1) = \{\{1,3\},\{0\},\{1\},\{1\}\}$. Cube $c_2(X_2)$ however can not be merged with any other cube. □

Cube merging can be used to compress a set of minterms or cubes and represent them by a smaller number of cubes without any loss of information. This property is an important one as the relation to be transformed into our representation is given in a form of MV cubes.

4. Labeled rough partitions

Definition 9 Separation of the elements of a nonempty set S into nonempty subsets S_i, $\bigcup S_i = S$, is called a **rough partition** (r-partition) of S. □

Definition of the rough partition allows the subsets S_i to be non-disjoint while the definition of a partition requires them to be disjoint.

Definition 10 (relation) Let S_1 and S_2 be sets. A **relation** R from S_1 to S_2 is a subset of Cartesian product $S_1 \times S_2$. A **relation** R **on** S_1 is a subset of $S_1 \times S_1$. □

Function is a special case of relation from S_1 to S_2 where every element $s_1 \in S_1$ is the first member of precisely one ordered pair $(s_1, s_2) \in S_1 \times S_2$.

Definition 11 (labeled partition block) Let $C(X)$ be a set of MV cubes, and relation R_k be defined by a cube $c_k(X_1), X_1 \subseteq X$, as follows: $c_i(X) R_k c_j(X)$ iff $c_k(X_1) \subseteq c_i(X_1)$ and $c_k(X_1) \subseteq c_j(X_1)$, where $c_k(X_1)$ is given and $c_i(X), c_j(X) \in C(X)$. The set of all cubes $c_i(X)$ being in relation R_k to each other and labeled by cube $c_k(X_1)$ will be called **labeled partition block** and denoted by $B_{c_k(X_1)}$. □

The labeled partition block consists of two elements: the partition block which is a set of cubes and the label which is a cube. Since all the cubes in $C(X)$ can be enumerated with distinct integer numbers, the partition block can be represented by a set of integer numbers. Any set of distinct symbols would work here as well. Label added to the partition block allows for establishing a correspondence between the set of numbers (or symbols) in the partition block and cubes in $C(X)$.

Definition 12 (labeled rough partition) The collection of all nonempty labeled partition blocks $B_{c_k(X_1)}$ will be called **labeled rough partition** (lr-partition) and denoted by $P(X_1) = \{B_{c_k(X_1)}\}$. □

In particular, if $X_1 = \{x\}$ then $P(X_1) = P(x)$, and, if $X_1 = X$ then $P(X_1) = P(X)$.

Example 7 Let $X = \{x_1, x_2, x_3, x_4\}$, $X_1 = \{x_1, x_3\}$, and $m_{x_1} = m_{x_3} = 2$. Then $P(X_1)$ will contain four (at most) blocks corresponding to the following $c_k(X_1)$ cubes: $00, 01, 10, 11$. In other words $P(X_1) = \{B_{00}, B_{01}, B_{10}, B_{11}\}$. □

Definition 13 For lr-partitions $P(X_1)$ and $P(X_2)$ of a set of cubes $C(X)$, $X_1, X_2 \subseteq X$, it is said that $P(X_1) \leq P(X_2)$ if every block of $P(X_1)$ is included in at least one block of $P(X_2)$: $\forall B_{c_i(X_1)} \exists B_{c_j(X_2)} \ni B_{c_i(X_1)} \subseteq B_{c_j(X_2)}$. □

Definition 14 (labeled partition block product) Product of two labeled partition blocks $B_{c_i(X_1)}$ and $B_{c_j(X_2)}$ is the labeled partition block $B_{c_k(X_3)} = B_{c_i(X_1)} \cap B_{c_j(X_2)}$, which partition block is an intersection of partition blocks of $B_{c_i(X_1)}$ and $B_{c_j(X_2)}$ and label $c_k(X_3)$ is equal to $c_i(X_1)\$c_j(X_2)$. □

Definition 15 (lr-partition product) The product $P(X_1)P(X_2)$ of lr-partitions $P(X_1)$ and $P(X_2)$ of a set of cubes $C(X)$, $X_1, X_2 \subseteq X$, is lr-partition $P(X_3), X_3 = X_1 \cup X_2$, the blocks of which are non empty products of the blocks of $P(X_1)$ and $P(X_2)$. □

Example 8 (lr-partition product) Let:
$X_1 = \{x_1, x_2\}, X_2 = \{x_3, x_4\}$,
$P(X_1) = \{B_{00}, B_{01}\}_{X_1} = \{\{0,1\}_{00}, \{2,3\}_{01}\}_{x_1x_2}$,
$P(X_2) = \{B_{00}, B_{11}\}_{X_2} = \{\{0,2\}_{00}, \{1,3\}_{11}\}_{x_3x_4}$.
Then:

$$\begin{aligned} P(X_3) &= P(X_1)P(X_2) \\ &= \{B_{0000}, B_{0011}, B_{0100}, B_{0111}\}_{X_3} \\ &= \{\{0\}_{0000}, \{1\}_{0011}, \{2\}_{0100}, \{3\}_{0111}\}_{x_1x_2x_3x_4} \end{aligned}$$

where $X_3 = X_1 \cup X_2$.

Theorem 1 *For any set of cubes $C(X)$, and any set of subsets X_i of X, $P(\bigcup_i X_i) = \prod_i P(X_i)$.*

PROOF It is enough to show that $P(X_1 \cup X_2) = P(X_1)P(X_2)$. By Definitions 11 and 12 lr-partition $P(X_i)$ consists of blocks corresponding to every combination of values of variables $x \in X_i$ present in data. Hence, by Definition 14 and 15 product $P(X_1)P(X_2)$ consists of blocks corresponding to every combination of values of variables $x \in X_1 \cup X_2$ present in data. Hence, $P(X_1 \cup X_2) = P(X_1)P(X_2)$. ■

Theorem 2 (lr-partition extraction) *For given sets of variables $X, X_1 \subseteq X$, and lr-partition $P(X) = \{B_{c_i(X)}\}$, lr-partition $P(X_1) = \{B_{c_j(X_1)}\}$ consists of partition blocks which are unions of those partition blocks of lr-partition $P(X)$ which labels meet the following condition:*

$$c_j(X_1) \subseteq c_i(X_1)$$

where $c_i(X_1)$ is that part of cube $c_i(X)$ which corresponds to variables $x \in X_1$.

PROOF Follows directly from Definitions 5, 11, and 12. ■

Example 9 (lr-partition extraction)
Let: $X = \{x_1, x_2, x_3\}, X_1 = \{x_2, x_3\}$ and:
$P(X) = \{B_{000}, B_{001}, B_{010}, B_{011}, B_{100}, B_{101}\}_{x_1x_2x_3}$
$= \{\{0\}_{000}, \{1\}_{001}, \{2\}_{010}, \{3\}_{011}, \{4\}_{100}, \{5\}_{101}\}_{x_1x_2x_3}$
Then: $P(X_1) = \{B_{00}, B_{01}, B_{10}, B_{11}\}_{x_2x_3}$ and:
$B_{00} = B_{000} \cup B_{100} = \{\{0\} \cup \{4\}\}_{00} = \{0,4\}_{00}$
$B_{01} = B_{001} \cup B_{101} = \{\{1\} \cup \{5\}\}_{01} = \{1,5\}_{01}$
$B_{10} = B_{010} = \{2\}_{10}$
$B_{11} = B_{011} = \{3\}_{11}$ □

5. Representation of MV relations

Theorem 3 (representation) *The MV, multi-output relation can be represented by a set of lr-partitions $\{P(X_1), \ldots, P(X_{n_x}), P(Y_1), \ldots, P(Y_{n_y})\}$, where $\bigcup X_i = X, \bigcup Y_i = Y$ and X, Y are sets of input and output variables respectively.*

PROOF It is enough to show that transition to another representation is possible. Transition to the cube representation can be performed by computing lr-partitions $P(X) = \prod_{i=1}^{n_x} P(X_i)$ and $P(Y) = \prod_{i=1}^{n_y} P(Y_i)$ and taking labels of $P(X)$ as input cubes and labels of $P(Y)$ as output cubes. The correspondence between input and output cubes can be determined in the following way: $c_i(X)$ corresponds to $c_j(Y)$ if $B_{c_j(Y)} \geq B_{c_i(X)}$. Since this a relation, one input cube may correspond to several output cubes. ■

Contrary to the rough partition [9] which stores an abstraction of a function, the labeled rough partitions can be used for general purpose representation of functions and relations because no information is lost in them.

Example 10 (representation) An example of MV, multi-output relation is shown in Table 1 and Figure 1. According to Theorem 3 relation can be represented in many different ways. Two of them are presented in Figure 1a and 1b.

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. MV multi-output relation. Rows correspond to MV cubes.

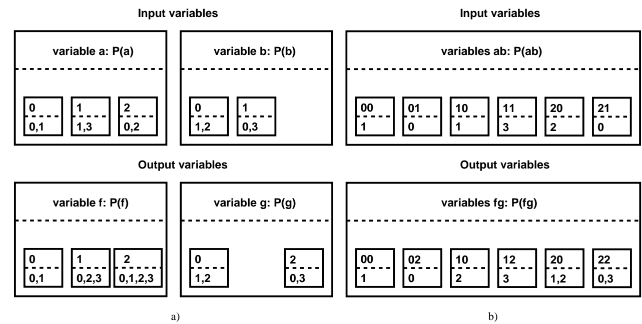

Figure 1. a) $P(X_i) = P(x_i)$**,** $P(Y_j) = P(y_j)$ **b)** $P(X_1) = P(X)$**,** $P(Y_1) = P(Y)$

In Figure 1, partition blocks correspond to the small squares, upper part of which contain block labels and the

lower part set of integer numbers, each number corresponding to one cube. Big squares correspond to labeled rough partitions for a given set of variables. Figure $1a$ shows lr-partitions $\{P(a), P(b), P(f), P(g)\}$ and Figure $1b$ lr-partitions $\{P(ab), P(fg)\}$. □

Lr-partition representation has been created to enable efficient representation of incompletely specified functions of many variables. If an (implicit) cube has standard output don't cares for **all** its outputs, it is not stored (explicitly) 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 that 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 lr-partition representation is at worst of the order of the number of cares, so it does not depend on the location of don't cares. In addition, for lr-partitions, the encoding of cubes with secondary variables decreases the size 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 lr-partitions to represent relations, the generalized value positions are stored in an efficient way, because they are treated in the same way as the input generalized values, and the sharing of subsets is used between all the variables. This is one more advantage of representing input and output variables uniformly.

6. Memory Requirements

The starting point for lr-partitions representation is a relation or function given in a form of MV cubes (see Table 2). Then lr-partitions are built for selected subsets of input and output variables (see Example 11). Memory requirement for lr-partition representation depends on two main factors:

1. Selection of sets X_i and Y_j the partitions are based on ([9] allows for $|X_i| = |Y_j| = 1$ only)
2. Representation of sets of cubes in the partition blocks

6.1. Selection of sets X_i, Y_i

Theorem 3 gives us a lot of freedom in selecting sets X_i and Y_i. Let us analyze memory requirements for such a data structure. Partition block can be considered an atomic data structure so the analysis will focus on minimizing the number of partition blocks required to represent a set of cubes. To simplify analysis let us assume that the number of values each variable can take is equal to m, each set X_i contains the same number of variables k, and sets X_i are disjoint. Then, for completely specified function represented by minterms, the number of blocks n_B is equal to:

$$n_B = \frac{n}{k} m^k$$

where $n = |X|$, $\bigcup X_i = X$, and X_i are disjoint.

In the above formula m, n are constants so $n_B = f(k)$ which takes minimum value for $k = 1/\ln m$. Calculation of k for different values of m results in $k = 1.44, 0.91, 0.72, \ldots$ for $m = 2, 3, 4, \ldots$. Since k has to be a natural number greater than 0 the best choice for k is $k = 1$ which leads to $\{P(x_1), P(x_2), \ldots, P(x_n), P(y_1), P(y_2), \ldots, P(y_k)\}$ in Theorem 3.

Situation is different, however, if the relation is incompletely specified and the number of care cubes is a small fraction of the number of all minterms specifying the relation (as it is often the case for ML data). For instance, if the number of care cubes grows linearly with the number of input variables, n_B will be described by the following formula:

$$n_B = \frac{n}{k} K n$$

where K is a constant.

The value of n_B in this equation decreases if the value of k increases. Since k can not be greater then n, n_B takes minimal value for k equal to n and the set of partitions in Theorem 3 reduces to $\{P(X), P(Y))\}$. In this case the number of blocks and their size (one element) are small but the storage required for labels (cubes) increases and becomes the primary factor determining memory requirement.

Between these two extreme cases there are many other possible choices. Selection of sets X_i and Y_i can also be done based on some heuristic measures of closeness of variables. Such operation would correspond to definition of a higher level abstraction and can easily by represented by lr-partitions. Example 11 illustrates two cases described above.

Example 11 An example of a MV relation is shown in Table 2 and can be represented as follows:
$P(x_1) = \{\{0,1,3\}_0, \{2,4\}_1\}_{x_1}$
$P(x_2) = \{\{0,3\}_0, \{1,2,4\}_1, \{3\}_2\}_{x_2}$
$P(x_3) = \{\{0,2,3\}_0, \{1\}_1, \{4\}_2\}_{x_3}$
$P(x_4) = \{\{0,1,2\}_0, \{1,3,4\}_1\}_{x_4}$
$P(y_1) = \{\{0,2,4\}_0, \{0,1\}_1, \{1\}_2, \{3,4\}_3\}_{y_1}$
$P(y_2) = \{\{0,3,4\}_0, \{0,1,2\}_1, \{0,2,3\}_2\}_{y_2}$

cube #	x_1	x_2	x_3	x_4	y_1	y_2
0	0	0	0	0	0,1	0,1,2
1	0	1	1	0,1	1,2	1
2	1	1	0	0	0	1,2
3	0	0,2	0	1	3	0,2
4	1	1	2	1	0,3	0

Table 2. MV relation

$$P(X) = \{\{0\}_{0\ 0\ 0\ 0}, \{1\}_{0\ 1\ 1\ 0,1}, \{2\}_{1\ 1\ 0\ 0}, \{3\}_{0\ 0,2\ 0\ 1}, \{4\}_{1\ 1\ 2\ 1}\}_{x_1 x_2 x_3 x_4}$$
$$P(Y) = \{\{0\}_{0,1\ 0,1,2}, \{1\}_{1,2\ 1}, \{2\}_{0\ 1,2}, \{3\}_{3\ 0,2}, \{4\}_{0,3\ 0}\}_{y_1 y_2}$$

Partitions $P(X), P(Y)$ represent relation with smaller number of blocks than partitions based on single variables. Labels are longer and more memory is needed to store them but the total memory requirement can still be lower then for single variable partitions.

6.2. Set representation

All MV operations on lr-partitions use *set-theoretical operations* on the corresponding sets of integer numbers 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 lr-partitions with no modification: for instance the packages for BMDDs, EVDDs, KFDDs, K*BMDs, ZBDDs, etc. [10]. We plan to compare the efficiency and storage requirements for lr-partitions with various data structures for partition blocks. Especially, we plan to experiment with these new packages in our lr-partition package. Currently we compared only two data structures: Bit Sets (BS), and U.C. Berkeley standard BDDs.

6.2.1 Binary Decision Diagrams (BDD)

One of the most efficient representations of a large set of objects is a decision diagram data structure, in particular *Binary Decision Diagram* (BDD), which has been very successfully applied to the representation of large binary functions [2].

A question of comparison of BDDs and cube arrays is a much-discussed one in logic synthesis [5, 17]. 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 [5] (or that occur in ML, logic or controller design [17]), that are more efficiently described using an array of cubes.

It is advantageous that with good selection of cube enumeration in these two extreme worst cases lr-partitions with blocks represented by BDDs **are comparable in size to the *better* representation of the two: arrays of cubes, or BDDs**.

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 lr-partition representation. Thus the size of the block for the ON-set of the output variable is the same as that of the BDD for this function. All the input blocks have one node each. Hence, both representations are comparable in space.

For the other extreme case, let us consider a binary function like those discussed by [5] that have polynomial number of cubes and exponential number of nodes in BDD. When the function is specified with cubes, it has n variables and k cubes, $k << 2^n$. Very conservatively estimating: in the worst case there is $2(n+1)$ partition blocks, each represented by a BDD with k nodes. So, the total number of nodes is $O(2nk)$ while the number of nodes in the single BDD representation 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. There exist practical functions with similar, although not that extreme properties [17]. To this category belong functions with many cubes and many variables, but with still very small ratio of cares to don't cares. This is the kind of functions from ML benchmarks, but with larger k, n and number of terms than in the functions from U.C. Irvine benchmarks. It is our hope that for larger multi-valued functions or relations the storage advantage of lr-partition representation will be even more clearly observable.

6.2.2 Bit Sets (BS)

Bit Set (BS), is an object that contains logically infinite set of bits. Because it is logically infinite, BS possesses a trailing, infinitely replicated 0 or 1 bit, called the "virtual bit", and indicated via 0* or 1*.

To represent any subset of a set of n elements, BS contains n bits maximum. If the i-th element of the set is contained in the subset, i-th bit of the BS is set to 1, otherwise it is set to 0. The memory requirement for this data structure depends on the enumeration of objects in the set, for instance $S_1 = \{0\}$ requires only two bits of memory (bit 0 equal to 1 and virtual bit 0*) while $S_2 = \{n-1\}$ requires $n+1$ bits, even though both sets contain only one element. Hence the maximum memory requirement for BS to represent any subset of a set of n elements is equal to $\lceil n/8 \rceil$ bytes.

7. Results

Notations lr-BDD and lr-BS will refer to lr-partition representations with BDDs and BSs representing partition blocks respectively. Notation BDD refers to representation of the function by a single BDD. Since lr-BS implementation was not ready at the time of writing this paper so the sizes of lr-BS are sizes computed according to the following formulae:

$$\text{size} = \left\lceil \frac{\text{\# of cubes} \cdot \text{\# of partition blocks}}{8} \right\rceil \text{[bytes]}$$

Where the number of partition blocks for binary function is double the number of both input and output variables (two blocks, labeled 0 and 1, per variable). To compute lr-BS/BDD we assume that one BDD node requires 22 bytes of memory [1]. All the tests were performed on DECstation 5000/240 with 64MB of memory. Times are user times measured with accuracy of 1/10 second by the Unix command `/bin/time` and are given in seconds. For lr-BDD and BDD representations we used U.C. Berkeley BDD package with sifting variable reordering method and lr-partitions based on single variables $\{P(x_1), P(x_2), \ldots, P(y_1), P(y_2), \ldots\}$.

7.1. Binary functions

The testing has been performed on three special classes of functions: parity, devadas, multiply, and on two level MCNC benchmarks. The reason for using two level MCNC benchmarks is that current implementation of lr-BDD accepts input data only in the form of multiple-valued cubes similar to Espresso format. This is a natural representation of input data in ML and controller design problems. The implementation which would accept multi-level description blif format is under development.

The examples analyzed in this section are completely specified binary functions (to allow for comparison with BDD representation). Lr-partitions however, allow for representation of multiple-valued functions and relations which can be incompletely specified.

7.1.1 Parity functions

For *parity* functions linearly-sized ($2n-1$ nodes) BDD representation can always be found, so in this case BDD should compare very good to other representations. The comparison of BDD and lr-BDD representations presented in Table 4 however, shows that lr-BDD is equally good for this type of functions and its size is equal to $2n$ nodes. The other representation however (lr-BS), compares poorly to both BDD and lr-BDD but it is not because the lr-BS representation is not good but because for this type of functions BDD performs extremely well.

	i/o	# cubes	t_{BDD} [s]	t_{lr-BDD} [s]
p9	9/1	512	0.3	0.6
p10	10/1	1024	0.6	1.6
p11	11/1	2048	1.5	3.6
p12	12/1	4096	3.7	8.3
p14	14/1	16384	19.2	40.8
p16	16/1	65536	97.1	194.8

Table 3. Parity functions: time

	lr-BDD nodes	BDD nodes	$\frac{\text{lr-BS}}{\text{BDD}}$	$\frac{\text{lr-BDD}}{\text{BDD}}$
p9	18	17	3.42	1.06
p10	20	19	6.74	1.05
p11	22	21	13.30	1.05
p12	24	23	26.31	1.04
p14	28	27	103.43	1.04
p16	32	31	408.40	1.03

Table 4. Parity functions: size

In terms of time, lr-BDD representation requires roughly twice as much time as BDD to read parity functions and that ratio remains constant when the function size increases.

7.1.2 Devadas functions

Another type of function to be tested was the one discussed in [5]. The function has $2n + \log n$ inputs and n^2 product terms in sum-of-product representation and $O(2^{n/2})$ nodes in BDD representation under any possible variable ordering. The functions d8, d10, d11, d12 in Tables 5 and 6 correspond to $n = 8, 10, 11,$and 12.

	i/o	# cubes	t_{BDD} [s]	t_{lr-BDD} [s]
d8	19/1	2038	23.3	12.8
d10	24/1	10308	251.2	104.8
d11	26/1	22631	831.6	249.9
d12	28/1	49151	2984.5	632.3

Table 5. Devadas functions: time

	lr-BDD nodes	BDD nodes	$\frac{\text{lr-BS}}{\text{BDD}}$	$\frac{\text{lr-BDD}}{\text{BDD}}$
d8	1743	1610	0.29	1.08
d10	3372	5331	0.55	0.63
d11	2403	16445	0.42	0.15
d12	11930	20784	0.78	0.57

Table 6. Devadas functions: size

In terms of memory requirement both lr-BS and lr-BDD are better than BDD. However, lr-BS memory requirement increases with the number of cubes and eventually may become greater than BDD. On the other hand, lr-BDD to BDD ratio decreases with the number of cubes, the larger the number of cubes the better lr-BDD comparing to BDD.

Similar situation is in terms of time needed to build a representation. Time $t_{lr_{BDD}}$ needed to build lr-BDD increases

slower than the time t_{BDD} needed to build a BDD and the ratio t_{BDD}/t_{lr-BDD} increases from 1.79 for d8 to 4.76 for d12.

The conclusion is that for devadas type functions lr-BDD is clearly the winner in terms of both memory requirement and time.

7.1.3 Multiplier functions

Another type of function is n-bit *multiplier* function which requires $O(2^{n/8}) = O(1.09^n)$ nodes in single BDD representation [3]. Functions m6, m7, m8, and m9 in Table 8 correspond to $n = 6, 7, 8$ and 9. As it can be seen from Table 8 the size of lr-BDD increases slower than that of BDD. This would indicate that the number of nodes of lr-BDD is less than $O(1.09^n)$.

In terms of time lr-BDD is roughly 1.5 times slower than BDD for multiply functions in Table 7.

	i/o	# cubes	t_{BDD} [s]	t_{lr-BDD} [s]
m6	12/12	4096	11.0	22.4
m7	14/14	16384	75.0	124.8
m8	16/16	65536	461.2	663.7
m9	18/18	262144	2419.5	3930.9

Table 7. Multiply functions: time

	lr-BDD nodes	BDD nodes	$\frac{\text{lr-BS}}{\text{BDD}}$	$\frac{\text{lr-BDD}}{\text{BDD}}$
m6	1109	1103	1.01	1.0050
m7	3116	3109	1.68	1.0020
m8	8849	8841	2.70	1.0009
m9	25063	25054	4.20	1.0004

Table 8. Multiply functions: size

7.1.4 MCNC benchmarks

The results of testing on MCNC benchmarks are shown in Tables 10 and 9. The lr-BS representation appears to be smaller than BDD representation in 73% of cases. The lr-BDD representation however, is larger than BDD in most of the cases.

For most of the functions in the table the number of cubes is much smaller then the number of minterms required to represent the same functions and, as the analysis in Section 6.1 shows, lr-partitions based on sets of variables ($P(X_i)$) instead of single variables ($P(x_i)$) should be less memory consuming here.

BDD representation for benchmarks `apex3` and `seq` failed to terminate successfully as it didn't fit into computer memory. The lr-BDD representation terminated without any problem for the same benchmarks. This would indicate that lr-BDD is less memory consuming when creating the representation. It can be explained by the fact that lr-BDD consists of many small shared BDDs which are incrementally created and processed while reading the data. On the

	i/o	# cubes	t_{BDD} [s]	t_{lr-BDD} [s]
apex1	45/45	1440	82.4	31.0
apex2	39/3	1576	42.2	31.0
apex3	54/50	1036	-	20.1
apex4	9/19	1907	1.9	13.8
apex5	117/88	2710	7.2	120.9
seq	41/35	2014	-	47.5
table3	14/14	1686	1.8	19.5
table5	17/15	1600	1.7	21.8
cps	24/109	855	3.2	10.3
cordic	23/2	2105	4.0	14.0
duke2	22/29	404	0.7	4.1
e64	65/65	327	3.5	5.0
ex1010	10/10	1297	2.4	9.7
ex4p	128/28	654	2.5	19.8
misex2	25/18	101	0.1	0.5
misex3	14/14	1391	4.0	15.5
misex3c	14/14	1566	2.3	16.3
pdc	16/40	822	1.6	7.9
spla	16/46	837	1.3	8.5
t481	16/1	841	0.7	4.5
vg2	25/8	304	0.9	2.8
alu4	14/8	1184	2.6	12.4
5xp1	7/10	141	0.0	0.4
9sym	9/1	158	0.1	0.6
bw	5/28	93	0.1	0.2
clip	9/5	271	0.2	1.3
ex5p	8/63	208	0.4	0.8
inc	7/9	94	0.0	0.3
rd53	5/3	67	0.0	0.1
rd73	7/3	274	0.1	1.1
rd84	8/4	511	0.2	2.3
sao2	10/4	137	0.1	0.5

Table 9. MCNC benchmarks: time

	lr-BDD nodes	BDD nodes	$\frac{\text{lr-BS}}{\text{BDD}}$	$\frac{\text{lr-BDD}}{\text{BDD}}$
apex1	4877	1345	1.09	3.63
apex2	5594	730	1.03	7.66
apex3	3384	-	-	-
apex4	4012	892	0.68	4.50
apex5	7435	1130	5.59	6.58
seq	6202	-	-	-
table3	4311	778	0.69	5.54
table5	4387	711	0.82	6.17
cps	2550	1072	1.21	2.38
cordic	2136	61	9.84	35.02
duke2	1364	392	0.60	3.48
e64	918	229	2.12	4.01
ex1010	3605	1314	0.23	2.74
ex4p	2952	535	2.17	5.52
misex2	369	78	0.65	4.73
misex3	4616	695	0.64	6.64
misex3c	3976	499	1.00	7.97
pdc	2820	609	0.86	4.63
spla	2598	576	1.03	4.51
t481	1368	32	5.12	42.75
vg2	1139	301	0.38	3.78
alu4	3478	800	0.37	4.35
5xp1	388	55	0.51	7.05
9sym	485	26	0.70	18.65
bw	228	105	0.34	2.17
clip	759	92	0.47	8.25
ex5p	562	242	0.69	2.32
inc	265	68	0.26	3.90
rd53	180	19	0.34	9.47
rd73	661	35	0.91	18.89
rd84	928	48	1.45	19.33
sao2	459	89	0.26	5.16

Table 10. MCNC benchmarks: size

other hand, BDD representation consists of one large DAG which may temporarily grow beyond capacity of the available memory while reading the data and performing necessary transformations.

7.1.5 Summary on binary functions

From the testing presented in the previous sections lr-BS appears to be less memory consuming than BDD on functions with relatively small number of cubes (MCNC benchmarks), lr-BS should also be faster than both lr-BDD and BDD (operations on bit sets are fast).

The other representation, lr-BDD, appears to be comparable or slightly larger then BDD in both size and time (with the exception of devadas functions). This is probably due to the structural difference of both representations. BDD consists of one large DAG while lr-BDD of many small BDDs. This structural difference is probably the reason why lr-BDD seems to be better suited than BDD to read and process large functions (MCNC benchmarks `apex3` and `seq`). More testing however is needed to fully verify this hypothesis.

7.2. Multiple-valued functions

Table 11 shows a comparison of selected benchmarks from U.C. Irvine ML repository, and from Prof. Bratko, Univ. of Ljubljana (`car, employ1, employ2, programm`) in terms of memory requirements for representation of partition blocks by BDDs and BSs. Value in column 4 (part blocks) corresponds to the total number of partition blocks.

	i/o	cubes	part blocks	nodes	lr-BS / lr-BDD	time [s]
zoo	16/1	101	46	412	0.07	0.4
shuttle	6/1	15	18	43	0.04	0.0
breastc	9/1	699	92	3638	0.10	5.3
balance	4/1	625	23	652	0.13	0.1
lenses	4/1	24	12	23	0.07	0.0
trains	32/1	10	107	98	0.10	0.0
trains20	29/1	20	109	185	0.08	0.1
car	6/1	1728	25	1163	0.21	4.5
employ1	7/1	18000	33	4292	0.79	26.4
employ2	9/1	9600	31	1802	0.94	66.3
programm	12/1	20000	47	70447	0.08	325.0

Table 11. ML benchmarks

As can be seen from Table 11 in all the cases lr-BS is smaller then lr-BDD, even for functions with a large number of cubes. However, the ratio lr-BS/lr-BDD not only depends on the number of cubes but also on the structure of the function. For instance, function `employ2`, which is much smaller then `programm`, has lr-BS/lr-BDD = 0.98, much larger the value then 0.08 for the `programm` function. If that ratio depended only on the number of cubes the relation would have been opposite.

8. Conclusions

We have presented a new data structure (lr-partitions) and shown that it gives very good results not only on binary functions but especially on a broader class of multiple-valued, completely and incompletely specified relations (functions in particular) which are typical in Machine Learning and complex FSM controller optimization applications.

By selecting suitable sets X_i, Y_i and partition block representation, lr-partitions can be tuned (optimized) for a given type of function or relation. This gives the designer a fast way of adjusting the representation for very large functions that would not fit into available computer memory (like `apex3` and `seq`). For instance lr-BDD based on single variables ($P(x_i)$) have characteristics similar to BDD representation (large partition blocks, small labels). If lr-BDD is based on larger sets ($P(X)$) then they more resemble cube representation of the function (small partition blocks, large labels).

Comparison of lr-partitions, with BS representing partition blocks, and single BDD shows superiority of lr-partitions in most of the binary test cases (73% of MCNC benchmarks) and all ML benchmarks. If the number of cubes describing a function or relation is large (tens of thousands) representing partition blocks with BDDs is usually less memory consuming than with BSs.

Characteristics of lr-partition representation can be summarized as follows:

- It can easily handle 'generalized values' to represent situations where both input (attribute value) and output (class assignment) variable values may be multiple. This is especially important in ML and complex FSM controller optimization applications to express uncertainty of choice of variable's values.
- It can easily handle situations where a variable is not present in a given cube (Michalski's train benchmark and '~' in Espresso format).
- By selection of sets X_i and Y_j lr-partitions can be dynamically adjusted to a given type of data (completely vs. incompletely specified, many cubes vs. few cubes) to minimize memory requirements.
- lr-partitions allow selection of set representation for partition blocks. Two such representations, BDDs and BSs, have been compared in this paper but other representations (OBDDs, BMDDs, EVDDs, KFDDs, hash tables, etc.) can be used too.

Based on lr-partitions, we implemented a decomposer of MV relations which can decompose large functions and relations from ML and controller domains. It proved that this

representation is not only compact but also allows for fast processing. We believe therefore that *Labeled Rough Partitions* are a new and very promising general purpose data structure for binary and MV functions and relations.

References

[1] K.S. Brace, R.L. Rudell, and R.E. Bryant. Efficient implementation of a BDD package. In *Proc. of 27th Design Automatiom Conference*, pages 40–45, June 1990.

[2] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *Trans. on Comput.*, C-35(8):667–691, 1986.

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

[4] A.L. de Oliveira. *Inductive Learning by Selection of Minimal Complexity Representations*. PhD thesis, University of California at Berkeley, 1994.

[5] S. Devadas. Comparing two-level and ordered binary decision diagram representations of logic functions. *IEEE Trans. on CAD*, 12(5):722–723, May 1993.

[6] S. Grygiel, M. Perkowski, M. Marek-Sadowska, T. Luba, and L. Jozwiak. Cube diagram bundles: a new representation of strongly unspecified multiple-valued functions and relations. In *Proc. of ISMVL'97*, pages 287–292, Halifax, Nova Scotia, Canada, May 28-30 1997.

[7] R. Kohavi. Bottom-up induction of oblivious read-once decision graphs. In *Europeean Conference on Machine Learning*, 1994.

[8] Y. T. Lai, K.R. Pan, M. Pedram, and S. Vrudhula. FGMap: A technology mapping algorithm for look-up table type FPGA synthesis. In *Proc. 30-th DAC*, pages 642–647, 1993.

[9] T. Luba. Decomposition of multiple-valued functions. In *Proc. 25th ISMVL*, pages 256–261, 1995.

[10] S. Minato. Graph-based representations of discrete functions. In *Proc. Reed-Muller'95 Workshop*, pages 1–10, Chiba, Japan, August 1995.

[11] M. Perkowski, T. Ross, D. Gadd, J. A. Goldman, and N. Song. Application of ESOP minimization in machine learning and knowledge discovery. In *Proc. Reed-Muller'95 Workshop*, pages 102–109, Chiba, Japan, August 1995.

[12] J.R. Quinlan. Induction of decision trees. *Machine Learning*, (1):81–106, 1986.

[13] T. Sasao. FPGA design by generalized functional decomposition. In T. Sasao, editor, *Logic Synthesis and Optimization*, pages 233–258. Kluwer Academic Publishers, 1993.

[14] V.Y. Shen, A. C. McKellar, and P. Weiner. An fast algorithm for the disjunctive decomposition of switching functions. *IEEE Trans. on Comput.*, C-20(3):304–309, March 1971.

[15] Shmerko, Jozwiak, and industry. private communication, 1996.

[16] A. Srinivasan, T. Kam, S. Malik, and R. Brayton. Algorithms for discrete function manipulation. In *IEEE International Conference on CAD*, pages 92–95, 1990.

[17] Steinbach, Hesse, Kempe, Rhode, and Barthel. Papers and discussions at the 2nd workshop boolesche probleme. Freiberg, Germany, 19-20 September 1996.

[18] Y.H. Su and P.T. Cheung. Computer minimization of multiple-valued switching functions. *IEEE Transactions on Computers*, C-21:995–1003, 1972.

[19] 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. In *Proc. Euro-DAC*, pages 230–235, 1992.