Implicit Representation of Discrete Objects

Alan Mishchenko
Portland State University
Department of Electrical and Computer Engineering, Portland, OR 97207
Tel: 503-725-2780, Fax: 503-725-3807
alanmi@ee.pdx.edu, http://www.ee.pdx.edu/~alanmi

Abstract

This tutorial paper discusses the known representations based on Binary Decision Diagrams (BDDs) for various types of discrete objects: incompletely specified functions, sets, finite state machines, binary and multi-valued relations, etc. While presenting the known material, the emphasis is on those aspects of implicit representations that are important to achieve speed-up in computation.

The new material includes implicit representations for dichotomies, partitions, set systems, and information measures. The last type of objects, information measures, constitute a promising approach to problem solving in a number of areas, including decomposition of discrete functions and finite state machines.

Multi-valued relations (MVRs) are presented as the most general representation for all the considered classes on discrete objects. Two complementary ways of representing MVRs are proposed: binary-encoded multi-valued decision diagrams (BEMDDs) and labeled rough partitions (LRPs). The sizes of BEMDDs and LRPs are compared using a set of multi-valued benchmarks.

1 Introduction

A wide range of scientific disciplines broadly grouped under the title "computer science and engineering" deal with various discrete objects: boolean functions, relations, combinatorial sets, partitions, finite state machines, etc. As the complexity of problems involving discrete data grows, the role of representations increases. It becomes more important than ever to have reduced storage size as well as faster and more diverse manipulation.

The last ten years are marked by the growing recognition of *Binary Decision Diagrams* (BDDs) as the representation of choice for discrete objects.

A short historical note might be applicable here. Research in decision diagrams was started in the 50's with the work of C.Y.Lee [1] and continued in the 70's by S.B.Ackers [2]. Building on their results, R.E.Bryant discovered efficient algorithms to perform operations on boolean functions represented by reduced decision diagrams [3]. The new algorithms operate directly on the graph structure of the operands and have worst case complexity proportional to the number of nodes in the representation graphs.

This discovery caused a virtual revolution in fields requiring massive processing of discrete information for two reasons. First, the graph-based representation gave rise to a more robust implementation of the known algorithms using BDDs as the main storage and computation engine. Second, a new generation of algorithms appeared, exploiting the graph structure to direct and speed-up computation. For instance, in one of the recently created approaches, the structure of BDDs is analyzed to determine good candidate components for bi-decomposition of boolean functions [4].

BDD-based algorithms are also known as *symbolic* or *implicit* algorithms. They are called "symbolic" because they represent relationships between various types of objects by introducing additional boolean variables that are later treated as parameters, labels, or symbols. BDD-based algorithms are called "implicit" to differentiate them from the earlier run of algorithms called "explicit". The difference between these two types of algorithms is described as follows:

"We say that a representation is *explicit* if the objects it represents are listed one by one internally. Objects are manipulated explicitly, if they are processed one after another. An *implicit* representation means a shared representation of the objects, such that the size of the representation is not linearly proportional to the number of objects in it. In an implicit representation many objects are processed simultaneously in one step" [5, p.37].

This tutorial assumes that the reader is familiar with the basic principles of boolean algebra and logic synthesis. The goal is to systematically introduce implicit representations for commonly used types of discrete objects, outline a range of potential applications, and give a list of references for further reading.

Section 1 covers the basic definitions used in the tutorial. Each of the following sections discusses different types of discrete objects in detail.

1 Basic Definitions

Reduced Ordered Binary Decision Diagrams (for simplicity called BDDs in the sequel) provide graph-based representations for completely specified boolean functions. The BDD for a non-constant boolean function is a directed acyclic graph. The *root* node of the BDD is located on top of the graph. Two *terminal* nodes labeled by constants 0 and 1 are located at the bottom. Other nodes (if they are present in the graph) constitute the set of *intermediary nodes*. Each non-terminal node of the diagram is labeled by a variable of the given function.

The diagrams are called "binary" because they represent functions over variables taking one of two values, 0 or 1. The diagrams are called "decision diagrams" because evaluation of a boolean function represented by the BDD consists in tracing the path from the root node to one of the terminal nodes. In each node belonging to the path, the decision is made to follow the branch labeled by 0 or 1 depending on the variable value. The 0-branch of a node is called *low edge*. The 1-branch is called *high edge*.

Fig. 1 shows a BDD for function F = ac + b. Here bold lines correspond to high edges and dashed lines to low edges.

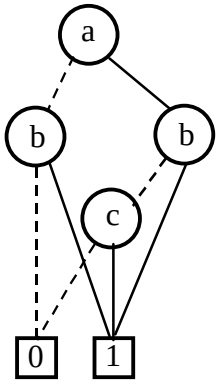

Fig. 1. The BDD for F = ac + b.

BDDs are called "ordered" because on any path variables appear in one particular order and no repetitions of variables on the path are allowed. For example, the BDD in Fig. 1 assumes variable order (a,b,c). The diagrams for other variable orders would look different and possibly have a different number of nodes.

The diagrams are called "reduced" because two *reduction rules* are applied when they are created:

(1) If several nodes are labeled by the same variable and have identical successors, only one of them is allowed to remain in the graph

(2) If both edges of a node have the same successor, the node is removed from the graph.

Assuming a fixed variable order and the above reduction rules, the resulting diagrams are *canonical*, that is, identical functions have identical diagrams and, vice versa, different functions have different diagrams.

The reader is referred to the following publications that provide an in-depth introduction into decision diagrams and proofs of the canonicity property [6,7].

In addition to the standard operations on boolean functions (AND, OR, EXOR, SHARP, etc.), BDD-based applications often use *quantification* over a variable and a subset of variables. *Existential (universal, unique)* quantification over variable x_i is the sum (product, exclusive sum) of cofactors of the given function with respect to x_i. This can be written as

$$\exists x_i F(x_1, x_2, ..., x_k) = F(\ldots x_{i-1},0,x_{i+1}\ldots) + F(\ldots x_{i-1},1,x_{i+1}\ldots)$$

$$\forall x_i F(x_1, x_2, ..., x_k) = F(\ldots x_{i-1},0,x_{i+1}\ldots) \,\&\, F(\ldots x_{i-1},1,x_{i+1}\ldots)$$

$$!x_i F(x_1, x_2, ..., x_k) = F(\ldots x_{i-1},0,x_{i+1}\ldots) \oplus F(\ldots x_{i-1},1,x_{i+1}\ldots)$$

A function after existential quantification has the following interpretation: it is true for the given assignment of variables if there exists a value of x_i (0 or 1) such that the original function is true for this value and the values from the given assignment. Similar interpretation can be given for other types of quantification.

Quantification over a set of variables is defined as the series of quantifications over each variable in the set. It can be proved that, for each type of quantification, the result does not depend on the order of variables being quantified.

The meaning of quantification can be illustrated using the Karnaugh map, in which the column variables are being quantified and the rows variables are all other variables. For such a map, the result of quantification is the function, whose map contains the same number of rows as the initial map but only one column (because the quantified variables no longer belong to the support of F).

The contents of this column depend on the type of quantification performed. In the case of existential quantification, the column is the sum of all the columns of the original Karnaugh map. Analogous observations hold for other types of quantification. Fig. 2 gives an example of existential and universal quantification of F= $\bar{a}$c +cd+abd with respect to the variable subset {a,b}.

ab	F				$\exists_{ab}F$	$\forall_{ab}F$
cd	00	01	11	10	cd	cd
00	0	0	0	0	00 0	00 0
01	0	0	1	0	01 1	01 0
11	1	1	1	1	11 1	11 1
10	1	1	0	0	10 1	10 0

Fig. 2. Example of existential and universal quantification

Existential quantification is also called *smoothing* because it can be done by crossing out quantified variables from the sum-of-products expression. In the above example

for the function F, removing literals $\bar{a}$, a, and b leads to $\exists_{ab}F(a,b,c,d) = c+d$, which is in agreement with Fig. 2.

2 Boolean Functions

As stated in the previous section, classical BDDs provide a flexible representation for *completely specified* boolean functions. Meanwhile, practical problems are often expressed using *incompletely specified* functions. One of the first attempts to represent incompletely specified functions was based on introducing a terminal node DC [8]. In such a modified diagram, the path corresponding to the assignment of variables, for which the function is a don't care, leads from the root to the new terminal node.

However, this approach did not become popular. Currently, there is a tendency to represent all types of discrete objects in terms of completely specified boolean functions, for which classical BDDs provide efficient storage and robust manipulation.

An incompletely specified boolean function can be represented by two completely specified functions. Some applications use the representation of ON-set and DC-set as BDDs. Other applications (including irredundant cover computation, two-level minimization and input support manipulation) rely on representation of function F as an interval: $F^1(x) \le F(x) \le F^{12}(x)$, there $F^1(x)$ is the ON-set and $F^{12}(x)$ is the sum of ON-set and DC-set.

Recently it was observed that decomposition algorithms are significantly simplified if an incompletely specified boolean function is expressed as a relation and represented using the characteristic function (introduced in Section 3). This approach constitutes a particular case of representation of multi-valued relations discussed in Section 9 and is mentioned here only for completeness of treatment of incompletely specified functions.

Given a boolean function F(x) with OFF-set $F^0(x)$, ON-set $F^1(x)$, and DC-set $F^2(x)$, F(x) can be represented using a completely specified function, R(x,v), over the set of input variables x and additional variable v as follows:

$$R(x,v) = \bar{v} \,\&\, F^0(x) + v \,\&\, F^1(x) + F^2(x)$$

3 Sets

A *set* is a non-ordered collection of elements of any kind. A set is perhaps the most fundamental mathematical abstraction and can be seen as a generic concept to build various representations of discrete objects. Therefore, it is important to represent sets compactly and perform set operations efficiently.

The key contribution to creating such a representation was made in 1977, when it was observed that an encoded set can be represented using the *characteristic function* [9]. From the mathematical point of view, the characteristic function of a logarithmically encoded set (an encoding that uses the shortest length code) is a completely specified boolean function, whose ON-set minterms stand in one-to-one correspondence with the elements of the given set.

The following is a more systematic definition of the characteristic function. Function $F: B^n \to B$, $B=\{0,1\}$, defines a subset of minterms of B^n, on which it is 1. Given a binary encoding of a set of elements, the characteristic function of the set is a boolean function, which is 1 for minterms that are used to encode the elements of the set and 0 for other minterms.

The only requirement for an encoding of elements is that boolean cubes representing different codes do not overlap. Expressed differently, it is equivalent to saying that the product of terms corresponding to two different codes is always zero.

For example, given a set of four elements $S = \{s_1,s_2,s_3,s_4\}$ and the encoding s_1=00, s_2=01, s_3=10, s_4=11, the subset $S_1=\{s_1,s_3,s_4\}$ can be represented by the characteristic function depending on variables x_1 and x_2:

$$\chi_{S1}(x_1,x_2) = \bar{x}_2\bar{x}_1 + x_2\bar{x}_1 + x_2x_1 = \bar{x}_1 + x_2$$

Characteristic functions represent sets efficiently because BDDs can be used to store and manipulate large completely specified functions. Set operations are reduced to operations on characteristic functions as follows:

Empty set: $\chi_{\varnothing} = 0$

Union of sets: $\chi_{S \cup T} = \chi_S + \chi_T$

Intersection of sets: $\chi_{S \cap T} = \chi_S \,\&\, \chi_T$

Difference of sets: $\chi_{S-T} = \chi_S \,\&\, (\chi_T)'$

Subset relation ($S \subset T$): $\chi_{S-T} = \chi_S \,\&\, (\chi_T)' = 0$

This representation turned out to be crucial for the success of numerous algorithms dealing with sets of FSM states, covers (sets of cubes) of boolean functions, sets of blocks of a partition or a set system, etc.

4 Sets of Subsets

To represent sets of subsets of elements of the given set, it is possible to *repeatedly* apply the technique introduced above for single sets. The set elements as well as sets themselves are logarithmically encoded. The resulting characteristic function depends on two groups of variables.

For example, given a set of four elements $S = \{s_1,s_2,s_3,s_4\}$ and encoding s_1=00, s_2=01, s_3=10, s_4=11, the problem is to find the characteristic function of the set of three subsets: $S_1=\{s_1,s_3,s_4\}$, $S_2=\{s_2\}$, $S_3=\{s_3,s_4\}$.

Suppose the encoding of subsets is S_1=(00), S_1=(01), and S_1=(10). Variables (x_2,x_1) are used to represent set elements and variables (y_2,y_1) are used to represent subsets themselves. Assuming an encoding of the least significant bit by the first variable, the following subset characteristic functions are derived:

$\chi_{S1} = \overline{x}_1 + x_2$

$\chi_{S2} = \overline{x}_2 x_1$

$\chi_{S3} = x_2\overline{x}_1 + x_2 x_1 = x_2$

The characteristic function of the set of subsets is

$\chi_{\{S1,S2,S3\}} = \overline{y}_2\overline{y}_1(\overline{x}_1 + x_2) + \overline{y}_2 y_1\,\overline{x}_2 x_1 + y_2\overline{y}_1\,x_2$

Figs. 3 and 4 show the Karnaugh map and the BDD for this function.

x_2x_1 \ y_2y_1	00	01	11	10
00	1	0	0	0
01	0	1	0	0
11	1	0	0	1
10	1	0	0	1

Fig. 3. The Karnaugh map for the given set of subsets

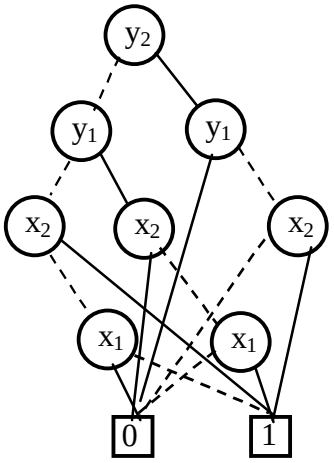

Fig. 4. The BDD for the given set of subsets

Positional encoding leads to another possibility to represent sets of subsets. In positional encoding, each element of the set is encoded using a separate variable. In the above example, assuming that variables (z_1,z_2,z_3,z_4) are used to encode the elements $\{s_1,s_2,s_3,s_4\}$, the following characteristic functions of the three subsets are derived:

$\chi_{S1} = z_1\overline{z}_2 z_3 z_4$

$\chi_{S2} = \overline{z}_1 z_2 \overline{z}_3 \overline{z}_4$

$\chi_{S3} = \overline{z}_1\overline{z}_2 z_3 z_4$

If a positional encoding is used, each subset is represented by one minterm of variables (z_1,z_2,z_3,z_4). The resulting characteristic function of the set of subsets is the sum of these minterms:

$\chi_{\{S1,S2,S3\}} = z_1\overline{z}_2 z_3 z_4 + \overline{z}_1 z_2 \overline{z}_3 \overline{z}_4 + \overline{z}_1\overline{z}_2 z_3 z_4$

Characteristic functions with positional encoding of elements are used in many problems involving manipulation of sets of sets, provided that the number of elements is not large (typically no more than 50 elements).

5 Cubes (Terms)

In this section, it is shown how BDDs can be used to efficiently implement operations on sets of boolean cubes (terms), i.e. arbitrary products of complemented and non-complemented boolean variables. This approach has been proposed in [10] to handle sets of primes and essential primes of incompletely specified boolean functions.

The performance of procedures using implicit cube representation is practically independent of the number of cubes in the set. It explains why this representation has played a central role in creating efficient procedures for irredundant cover computation [11] and two-level sum-of-products minimization [12], which outperformed the widely known explicit SOP minimizer ESPRESSO [13] by orders of magnitude on large examples.

Suppose P^n is the set of all cubes that can be constructed using variables $(x_1,x_2,...,x_n)$. The set P^n has 3^n elements, so at least $\lceil \log_2(3^n) \rceil$ Boolean variables are needed to represent any element and any subset of P^n. However, using this optimal encoding, there is no direct link between an element of P^n and its representation. In order to establish a direct link, it has been suggested to use 2*n variables, called *signature (polarity) variables* and denoted $(s_1,s_2,..,s_n)$ and $(p_1,p_2,..,p_n)$.

A mapping from the set of $\{0,1\}^n \times \{0,1\}^n$ into the set P^n is defined in the following way: a signature variable is zero if and only if (iff) a variable is not included in the cube; a polarity variable is 0 if the variable has a negative polarity in the cube and 1 if the variable has a positive polarity. For example, couple {[1101], [10-1]} denotes the cube $x_1\,\overline{x}_2\,x_4$.

In general, the couple (s, p) of $\{0,1\}^n \times \{0,1\}^n$ denotes a unique product of P^n standing for a unique subset of the domain $\{0,1\}^n$ encoded using variables $(x_1,x_2,...,x_n)$. The characteristic function connecting the variables s, p, and x is

$$\chi_T(s, p, x) = \prod_{k=1}^{n} (s_k \Rightarrow (x_k \Leftrightarrow p_k))$$

The predicate ÷ expresses the property that cubes t and t′ represented by couples c = (s, p) and c′= (s′, p′) are identical, which is true iff they consist of the same literals.

$$t \div t' = \prod_{k=1}^{n} ((s_k \Leftrightarrow s'_k) \,\&\, (s_k \Rightarrow (p_k \Leftrightarrow p'_k)))$$

The predicate ≥ expresses the property that cube t represented by the couple c = (s, p) contains cube t′ represented by the couple (s′, p′), which is the case iff all literals of t′ also occur in t:

$$t \geq t' = \prod_{k=1}^{n} (s_k \Rightarrow (s_k' \,\&\, (p_k \Leftrightarrow p_k')))$$

The variable ordering that minimizes the BDDs of these predicates and the computational cost of the cube handling procedures is given in [10]:

$s_1 < s_1' < p_1 < p_1' < \ldots < s_n < s_n' < p_n < p_n'$.

Here is an example of the use of the above representation to derive the characteristic function of the set of all cubes belonging to the ON-set of completely specified function F(x).

Proposition 1. The characteristic function of the set of all products that are cubes of F can be computed as follows:

$$\chi_C(s,p) = \exists x(\chi(s, p, x)) \,\&\, \forall x(\chi(s, p, x) \Rightarrow F(x)) .$$

The proof of this proposition follows from definitions of the characteristic function and the set of products.

Proposition 2. To get the set of all primes of the given function, the covered cubes should be eliminated from the above cube set represented by its characteristic function. It can be achieved by the following transformation:

$$\chi'_P(s,p) = \forall s'p'([\chi(s', p') \,\&\, (s', p') \geq (s, p)] \Rightarrow (s', p') \div (s, p))$$

The proof follows from definitions of the prime of the given function and predicates ≥ and ÷.

6 Finite State Machines and Finite Automata

First, relevant definitions are given from [14].

A *deterministic finite automaton* (DFA) is a quintuple $A = (K, \Sigma, \delta, q_0, F)$, where K is a finite set of states, Σ an alphabet, δ the transition function, $\delta: K \times \Sigma \rightarrow K$, $q_0 \in K$ the initial state, and $F \subseteq K$ the set of final states.

A *non-deterministic finite automaton* (NFA) is a quintuple $A = (K, \Sigma, \delta, q_0, F)$, where the transition relation δ is a mapping $\delta: K \times \Sigma^* \rightarrow 2^K$, and Σ* is the set of strings obtained by concatenating zero or more strings from Σ.

A string is *accepted* by A if it drives A into one of its final states. The language accepted by A, L(A), is the set of strings, which A accepts.

A *finite state machine* (FSM) is a six-tuple $M = (I, O, Q, \delta, \lambda, q_0)$ where I is a finite input alphabet, O a finite output alphabet, Q a finite set of states, δ the transition function, λ the output function, and q_0 the initial state.

A machine is of *Moore type* if λ does not depend on the inputs, and *Mealy type* otherwise. An FSM can be represented by a state transition graph (STG). A machine, in which transitions under all input symbols from every state are defined as a *completely specified machine*. In a completely specified machine, both δ and λ are completely specified functions. Otherwise, a machine is *incompletely specified*.

When BDDs are used to represent FAs and FSMs, their symbolic inputs, outputs (for FSMs), and current states are encoded using binary codes. The encoding may be arbitrary, except that codes should be disjoint. In practice, the size of BDDs is reduced if an adjacent (Grey) code is used for as many pairs of adjacent states (states connected by a transition) as possible.

Once the encoding is selected, the transition behavior and output behavior (for FSMs) can be specified as binary relations.

Thus, the transition behavior of the state machine with symbolic input I and two states, S_0 and S_1, is a binary relation T(i,x,y) over variables i, x, and y encoding the input, the current state, and the next state. The truth table and the BDD of T(i,x,y) are shown in Figs. 5 and 6.

I	CS	NS	i	x	y	T
a	S_0	S_0	1	0	0	1
b	S_0	S_1	0	0	1	1
a	S_1	S_1	1	1	1	1
b	S_1	S_0	0	1	0	1
			other codes			0

Fig. 5. Example of an FSM.

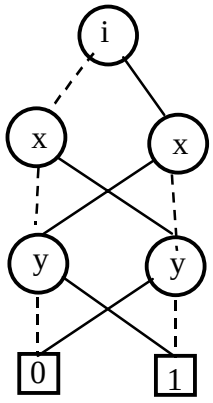

Fig. 6. BDD for the transition relation of FSM in Fig. 5.

The implicit representation carries complete information about the transition behavior of the FSM given by the state transition graph. For example, the transition under input b (encoded by 0) from state S_0 (encoded by 0) to state S_1 (encoded by 1) corresponds to the assignment of BDD variables to 001. Notice, for example, that the path corresponding to the assignment of variables i = 0, x = 0, and y = 1, leads from the root to the terminal node 1.

If an FSM is initially specified by an STG, the transition relation is computed by adding (OR-ing) the cubes corresponding to individual transitions. Each of these cubes is the product of smaller cubes, depending on variables that are encoding inputs, current states, and next states. In the above example, the transition relation is the sum of four cubes encoding transitions in Fig. 6:

$$T(i,x,y) = i\bar{x}\bar{y} + \bar{i}\bar{x}y + ixy + \bar{i}x\bar{y} = i \oplus x \oplus y$$

If the FSM is initially specified by the sequential net list, it is possible to first extract the STG and next derive the transition relation from the STG, as described above. However, the STG received at the intermediary stage of this process may be too large. A more efficient approach is possible. First, the next state functions, $\delta_k(i, x)$, depending on the input variables i and current state variables x, are derived by traversing the net list. The transition relation is computed using the formula

$$T(i,x,y) = \prod_{k=1}^{n} [\, y_k \Leftrightarrow \delta_k(i, x)\,]$$

where the product is the boolean AND operator and $\Leftrightarrow$ stands for the boolean equivalence operator ($a \Leftrightarrow b$ is defined as $\bar{a}\bar{b} + ab$). The transition function in the above example is

$$\delta(i,x) = \bar{i}\bar{x} + ix\ .$$

Hence, the expression for T(i,x,y) is computed as follows:

$$T(i,x,y) = \bar{y}\ \overline{\delta(i, x)} + y\,\delta(i,x) = \bar{y}\,(i\bar{x} + \bar{i}x) + y(\bar{i}\bar{x} + ix) =$$

$$= i\bar{x}\bar{y} + \bar{i}\bar{x}y + ixy + \bar{i}x\bar{y}$$

The reverse transformation also can be performed. Given the transition relation, the next-state functions are derived using the formula:

$$\delta_k(i,x) = \exists y\,[T_{y_k=1}(i, x, y)]$$

where the expression in square parentheses is the positive cofactor of the transition relation with respect to the next state variable y_k.

In a similar way, it is possible to derive the output relation, O(i,x,o), from the set of output functions, $\lambda_k(i,x)$, and vise versa.

A wide range of practical applications, in particular those dealing with verification and sequential equivalence checking, rely on implicit state enumeration, or reachability analysis, for finite-state machines [15,16]. During reachability analysis, the states of the FSM are visited in the breadth-first manner starting from the set of initial states. The computation of the reachable states can be performed efficiently even for very large FSMs due to the well-developed methodology of handing transition relations using BDDs in the way described above.

The set of FSM states reachable in one transition from the given set is computed as an image of the given set with respect to the transition relation as follows:

$$R(y) = \exists y \exists i\,[T(i, x, y)\ \&\ S(x)]$$

All the state sets in this procedure are presented as their characteristic functions using BDDs. This computation (known also as the *image computation*) is iterated until no new states can be reached. The set of reachable states is created as the union of the initial state set and state sets received during successive iterations.

The reachable state information can be used in a number of ways, for example, to simplify the transition and output relations by restricting them to those states that are reachable from the initial ones. Assuming that $A_R(x)$ is the characteristic function of the reachable states, the simplified transition relation is computed as follows:

$$T_S(i,x,y) = T(i,x,y)\ \&\ A_R(x)\ \&\ A_R(y).$$

Besides reachability analysis, a wide range of practical problems dealing with state machines can be solved using the representation of FSMs in terms of the transition relation and the output relation. For example, the state machine given by the transition relation is input-completely-specified iff

$$\forall x \exists y\,[T(i, x, y)]\ \&\ \forall x \exists o\,[O(i, x, o)] \equiv 1.$$

Other problems that can be solved include but are not limited to FSM state minimization [17,18,19,20,21], determinization, simplifying the networks of state machines [x, 22], , FSM decomposition, sequential ATPG [23], etc.

7 Partitions, Dichotomies, Set Systems

7.1. Partitions

The following definitions are taken from [24].

Partition π on a set of elements S is a collection of disjoint subsets of S, whose set union is S, i.e.

$$\pi = \{\, B_a \} : B_a \cap B_b = \varnothing,\ \forall a \neq b;\ \cup B_a = S.$$

The subsets of π are called *blocks* of π. The block containing s is designated by $B_\pi(s)$. If two elements s and t belong to the same block, $B_\pi(s) = B_\pi(t)$, it is written $s \equiv t(\pi)$.

The following two operations on partitions are used to define the ordered set of all partitions called the *lattice*, which plays a prominent role in practical applications, in particular, those dealing with the decomposition of finite state machines.

The *product* of partitions π_1 and π_2 on S is the partition $\pi_1 \cdot \pi_2$ on S such that $s \equiv t(\pi_1 \cdot \pi_2)$ iff $s \equiv t(\pi_1)$ and $s \equiv t(\pi_2)$.

The *sum* of partitions π_1 and π_2 on S is the partition $\pi_1 + \pi_2$ such that $s \equiv t(\pi_1 + \pi_2)$ iff there is a sequence in S, $s = s_0, s_1, \ldots, s_n$, such that $s_n = t$ and either $s_i \equiv s_{i+1}(\pi_1)$ or $s_i \equiv t_{i+1}(\pi_2)$, $0 \le i \le n\text{-}1$.

A *relation* between sets S and T is a subset R of $S \times T$:

$R = \{\, (s,t) \mid t = R(s),\ s \in S,\ t \in T \,\}$

A relation R on $S \times S$ is

reflexive if $\forall s \in S,\ s = R(s)$,

symmetric if $t = R(s)$ implies $s = R(t)$,

transitive if $t = R(s)$ and $u = R(t)$ implies $u = R(s)$.

A relation on $S \times S$ is called an *equivalence relation* if it satisfies all three of the above properties: it is reflexive, symmetric, and transitive.

If R is an equivalence relation on S × S, and s is an element of S, then the set $B_R(s) = \{ t \mid t \in S, t = R(s) \}$ is an *equivalence class defined by s*.

The following property is used to create an implicit representation of partitions:

Property. If R is an equivalence relation on S, the set of equivalence classes defines a partition π on S, and, conversely, a partition π on S defines an equivalence relation R on S whose equivalence classes are blocks of π. Thus if R defines π, then t = R(s) iff $s \equiv t(\pi)$.

To create an implicit representation of partitions, the elements of S are encoded using arbitrary disjoint codes. For many practical purposes, it is better to use the logarithmic encoding when the number of encoding variables is minimum, $\lceil \log_2|S| \rceil$.

Given the binary encoding of elements, a partition can be seen as a binary relation R. A pair of codes representing elements s and t are related through R iff they belong to the same partition block. For the codes that are not used in the encoding, the relation is defined as zero.

For example, given a set of three elements $S = \{s_1,s_2,s_3\}$ and the encoding $s_1 = (01)$, $s_2 = (10)$, $s_3 = (11)$, the problem is to build the binary relation representing the partition $\pi = \{ \{s_1,s_3\}, \{s_2\} \}$. The relation connects the following pairs of elements: for the first block (s_1,s_1), (s_3,s_3), (s_1,s_3), (s_3,s_1), for the second block (s_2,s_2), and does not connect any other elements. Therefore, the value of function F in Fig. 7 is zero for all other pairs of symbols s_i and s_j.

E_1	E_2	C_1	C_2	F
s_1	s_1	01	01	1
s_1	s_3	10	11	1
s_3	s_1	11	10	1
s_3	s_3	11	11	1
s_2	s_2	10	10	1
other pairs				0

	00	01	11	10
00	0	0	0	0
01	0	1	0	0
11	0	0	1	1
10	0	0	1	1

Fig. 7. Relation and its characteristic function

Function F can be represented as a completely specified function over two couples of variables that encode elements of E_1 and E_2. This completely specified function can be represented using BDD.

Operations of partitions are reduced to operations on their characteristic function represented using BDDs similar to how operations on sets are preformed by handling their characteristic functions. To find the product of partitions, the product of characteristic functions is computed. To find the sum of partitions, the sum of their characteristic functions is found followed by the computation of the transitive closure.

For example, the product of two partitions $\pi_1=\{ \{s_0\}, \{s_1,s_2,s_3\} \}$ and $\pi_1=\{ \{s_0,s_1\},\{s_2,s_3\} \}$ on the set of four elements $S = \{s_0,s_1,s_2,s_3\}$ with straight binary encoding can be computed as the product of their characteristic function Ψ_{π_1} and Ψ_{π_2}, shown in Fig. 8

Ψ_{π_1}

	00	01	11	10
00	1	0	0	0
01	0	1	1	1
11	0	1	1	1
10	0	1	1	1

Ψ_{π_2}

	00	01	11	10
00	1	1	0	0
01	1	1	0	0
11	0	0	1	1
10	0	0	1	1

$\Psi_{\pi_1 \cdot \pi_2}$

	00	01	11	10
00	1	0	0	0
01	0	1	0	0
11	0	0	1	1
10	0	0	1	1

Fig. 8. Example of computing the product of partitions.

7.2. Dichotomies

Dichotomy is a two-block partition of elements, whose blocks do not necessarily contain all the elements of the set. As such a dichotomy is very similar to a partition and can be represented in the same way.

7.3. Set Systems

Next, a generalization of partitions is considered. This discrete object plays an important role in decomposition of boolean functions and state machines [24].

A collection of subsets $\varphi = \{ B_a \}$ of S is called a *set system* (blanket) iff:

1) the union of all subsets is S ($\cup B_a = S$)
2) none of the subsets is completely contained in another subset ($B_a \subseteq B_b$ implies a = b)

Operations on set systems are introduced using the *MAX-operator* that takes $\{S_i\}$, a set of subsets of the set S. The result of applying the MAX-operator is the set of subsets that are not contained in any subsets of $\{S_i\}$:

$$MAX\{S_i\}=\{ B \subseteq S \mid B \in \{S_i\} \text{ and } S_k \subseteq B, S_k \in \{S_i\}, \text{implies } S_k = B \}$$

The product and sum of set systems are the following:

$$\varphi_1 \cdot \varphi_2 = MAX\{ B_1 \cap B_2 \mid B_1 \in \varphi_1 \text{ and } B_2 \in \varphi_2 \}$$

$$\varphi_1 + \varphi_2 = MAX\{ \varphi_1 \cup \varphi_2 \}$$

The characteristic function, F, of a set system is defined similarly to that of a partition. Given a binary encoding of the set elements, a pair of codes representing elements s and t constitute a minterm of F iff they belong to the same block of the set system.

Operations on set systems are reduced to operations on their characteristic functions. The peculiar property of this representation is that the product and the sum operations on sets systems are reduced to boolean operations on their characteristic functions, and are even simpler than that for partitions! The product and sum of partitions are computed as the boolean product and sum of their characteristic functions, without the need to compute transitive closure, which, computationally, is a relatively expansive operation.

Another advantage of this representation of set systems is that, by introducing additional labeling variables, it allows us to create characteristic functions of set system

pairs, sets of set systems and sets of set system pairs. These functions can store thousands of objects of the above kind and perform operations on them implicitly.

For example, using the above representation it is possible to create a BDD-based algorithm that derives all partitions satisfying substitution property [24]. Next, a formula with quantifiers can be used to find pairs of these partitions that lead to parallel FSM decomposition, etc.

8 Information Measures

According to [24], it is possible to "approximate the vague concept of information by the precise concept of a partition, i.e. information ≈ partition". However, it was not until recently that the precise concept of information relationships and measures have been introduced as an "analysis apparatus for efficient information system synthesis" [25].

Information measures draw upon the concept of *elementary (atomic) information*, that is, information necessary to distinguish between two elements, s_i and s_j, of the set S. Information contained in a partition or set system can be represented by the *information set*, which is defined as relation I on $S \times S$:

$$I = \{ \{s_i, s_j\} \mid s_i \text{ is distinguished from } s_j \}$$

The following definition of the information set is given in [25]: "IS (information set) contains the pairs of symbols that are not contained in any single block of a corresponding SS (set system)". Given the characteristic function of the set of elements, A(x), and the characteristic function of the set system (partition), $\Psi_\varphi(x, y)$, the characteristic function of the information set, $\Psi_{IS}(x, y)$ can be derived as follows:

$$\Psi_{IS}(x, y) = [\Psi_\varphi(x, y)]' \,\&\, A(x) \,\&\, A(y)$$

As before, the ON-set minterms of $\Psi_{IS}(x, y)$ are codes x and y of elements s_i and s_j that are distinguishable using the information represented in the given set system.

If information sets are represented as shown above, various operations of information measure theory can be efficiently implemented using elementary boolean operations on BDD of the characteristic functions. For example, common information is computed as a product of characteristic functions; extra information is computed as a set difference of characteristic functions, etc.

Information similarity measure and information increase measure can be computed as ½ of the number of minterms in the characteristic functions of the common information and extra information. In some applications, there is a need to compute a number of information measures (for example, to evaluate the informational contribution of each variable). A group evaluation of information measures can be efficiently implemented in such a way that each node of the shared ROBDD representation of the set of characteristic functions is visited at the most once.

9 Multi-Valued Relations

The above consideration of various discrete objects leads us to consider the most general and complex type of discrete objects, multi-valued relations (MVRs). Other types of discrete objects can be seen as particular cases of a multi-valued relation over a finite set of variables over finite domains.

For example, the characteristic function of a set of subsets is an MVR over two multi-valued variables, one of them having as many values as there are subsets, another as many values as there are elements in the subset. Given a positionally encoded set of elements, the MVR takes the form of a relation over as many binary variables as there are elements in the initial set. This relation is true for those assignments of the variables, which stand for the subsets.

As another example, consider the general case of a finite state machine before encoding. The transition relation of this state machine can be seen as a relation over three multi-valued variables: input, current state, and next state variables. After encoding, this machine becomes a relation over sets of binary variables encoding the multi-valued variables, which, again, is a particular case of an MVR.

The above discussion leads to the conclusion that it is important to have a good representation for MVRs. In this paper, we advocate the use of binary-encoded multi-valued decision diagrams (BEMDDs) as the representation of choice for MVRs. The motivation is that BEMDDs are efficient for large functions and lead to improved manipulation procedures [26], as compared to other known representations of MVRs: multi-valued cubes and cube partitions [27], edge-valued BDDs [28], classical BDDs [29], and classical MDDs [30].

In our approach, multi-valued variables are encoded using the smallest possible sets of binary variables. Thus, a k-valued variable requires at least $\lceil \log_2(k) \rceil$ binary variables to uniquely encode all its values. (Here the vertical bars stand for the closest larger integer.)

For example, a 5-valued variable A can be encoded using the set of three binary variables $\{a_1,a_2,a_3\}$. In the simplest case, the set of all possible values of variable A $\{0,1,2,3,4\}$ is encoded using the set of binary cubes:

$$\{ \bar{a}_3\bar{a}_2\bar{a}_1, \bar{a}_3\bar{a}_2a_1, \bar{a}_3a_2\bar{a}_1, \bar{a}_3a_2a_1, a_3\bar{a}_2a_1 \}$$

If k is not an integer power of two, this binary encoding results in $2^{\lceil \log 2(k) \rceil}$ - k spare minterms. It is possible to leave them unused and account for them in the decomposition routines. In this case, it is necessary to remember that the domain of binary variables encoding input variables is limited only to those vertices that provide codes for the values of multi-valued variables. This, however, increases

the BEMDD size and makes traversal routines more complicated.

From the practical point of view, it is better to distribute the unused minterms between the values of the function. For example, consider the two encodings of ten values using four binary variables shown in Fig. 9. Minterms of these tables stand for the values given in each cell.

	00	01	11	10
00	0	4	8	8
01	1	5	9	9
11	3	7	10	10
10	2	6	10	10

	00	01	11	10
00	0	2	7	4
01	0	2	8	4
11	1	3	9	6
10	1	3	10	5

Fig. 9. Two different ways to encode ten values using four binary variables

Currently, as the default encoding of values we use the encoding shown on the left of Fig. 9 and continue to experiment with other encodings.

Because outputs of MVRs are typically multi-valued, the outputs are represented using an encoding scheme similar to the inputs. Variables used for this purpose are called *output-value-encoding variables* (or simply, *value variables*). Traversal algorithms typically require ordering of the value variables below other variables in the BEMDD.

When the input and output multi-valued variables are encoded, the BEMDD representing an MVR can be constructed as a binary relation. This relation connects encoded input values with the corresponding encoded output values. For example, shown in Fig. 10 is a three-valued-output MVR over binary variable A and ternary variable B. Using the above approach, this function is represented as a binary relation over five binary variables (three variables for inputs and two variables for the output).

B\A	0	1
0	-	2
1	1	-
2	1,2	0,1

Fig. 10. The truth table for MVR F(A,B)

Suppose binary variable a encodes multi-valued variable A, variables b_1 and b_2 encode B, while variables v_1 and v_2 encode output values of F(A,B). The variable with the lower index corresponds to the less significant bit in the following encoding (0,1,2) = (00,01,1-). The characteristic function of the MVR F(A,B) can be expressed using variables (a,b_1,b_2,v_1,v_2) as follows:

$$R(a,b_1,b_2,v_1,v_2) = \bar{a}\bar{b}_2\bar{b}_1 + \bar{a}\bar{b}_2 b_1\bar{v}_2 v_1 + \bar{a}b_2(\bar{v}_2 v_1 + v_2) + \\ + a\bar{b}_2\bar{b}_1 v_2 + a\bar{b}_2 b_1 + ab_2\bar{v}_2 .$$

Each of the six terms is obtained directly by encoding a cell of the map in Fig. 10. For example, cube $\bar{a}\bar{b}_2\bar{b}_1$ stands for the cell corresponding to $A^{\{0\}}B^{\{0\}}$, cube $\bar{a}\bar{b}_2 b_1\bar{v}_2 v_1$ stands for $A^{\{0\}}B^{\{1\}}$, etc. The Karnaugh map and the BEMDD for $R(a,b_1,b_2,v_1,v_2)$ are shown in Figs. 11 and 12.

$v_2v_1 \setminus ab_2b_1$	000	001	011	010	110	111	101	100
00	1	0	0	0	1	1	1	0
01	1	1	1	1	1	1	1	0
11	1	0	1	1	0	0	1	1
10	1	0	1	1	0	0	1	1

Fig. 11. The Karnaugh map for MVR F(A,B)

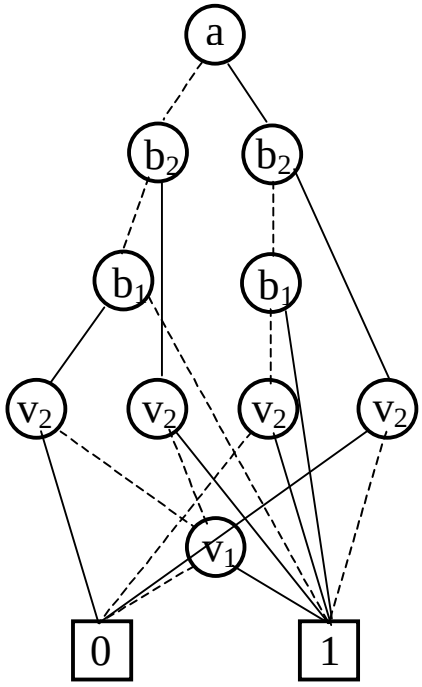

Fig. 12. The BEMDD for MVR F(A,B)

10 Labeled Rough Partitions

Recently, *Labeled Rough Partitions* (LRPs) have been introduced as a new representation of MVRs [31,32]. This representation is similar to BEMDDs in that it relies on encoding and characteristic functions. It is different from BEMDD representation in that the cube table specifying the MVR is read and encoded vertically, column-by-column, rather than horizontally, line-by-line.

Given the cube table specifying an MVR, the LRP is the set of subsets of rows corresponding to particular values of input and output variables. The idea of LRPs is best understood from an example. Fig. 13 shows the cube table for the MVR used as an illustration in the previous section.

	A	B	F
s_1	0	1	1
s_2	0	2	1,2
s_3	1	0	2
s_4	1	2	0,1

	A^0	A^1	A^2	B^0	B^1	B^2	F^0	F^1	F^2
s_1	+				+			+	
s_2	+					+		+	+
s_3		+		+					+
s_4		+				+	+	+	

Fig. 13. The cube table and LRP for MVR F(A,B)

Input cubes corresponding to don't-cares ($A^{\{0\}}B^{\{1\}}$ and $A^{\{1\}}B^{\{1\}}$) are not included in the table. To the left of the table there is a column of symbols representing table rows.

An LRP for relation F can be constructed as follows. For each value A, B, and F, create a set of symbols corresponding to the rows that contain these symbols. These sets are marked using "+" in the table given to the right of the cube table for F(x). Next, symbols $\{s_1,s_2,s_3,s_4\}$, multi-valued constants {0,1,2} and input/output variables of the relation {A,B,F} are encoded using three sets of binary variables (x_2,x_1), (y_2,y_1), and (z_2,z_1). (Notice that LRPs do not differentiate between inputs and outputs of the relation.)

The BDD-based representation of the LRP is the characteristic function over the encoding variables, which is true for those assignments of variables (x_1,x_2,y_1,y_2,z_1,z_2) that correspond to symbols belonging to particular values of given variables. Assuming the encoding of symbols (s_1,s_2,s_3,s_4) = (00,01,10,11), of ternary values (0,1,2) = (00,01,1-), and of input variable A=(00), the contribution of this variable to the characteristic function of the LRP is

$$\chi_{LRP}^{A} = \overline{z}_2\overline{z}_1[\ \overline{y}_2\overline{y}_1\,\overline{x}_2 + \overline{y}_2 y_1\, x_2\ ]$$

Similarly, it is possible to compute the contributions of the input variable B and the output variable F.

Table 1 gives the comparison of the node size and computation time for BEMDDs and LRPs for the set of POLO benchmarks [33]. The measurements were made on Pentium 266MHz computer with 64Mb of RAM.

The dash in the table means that we could not build the BEMDD for the largest benchmark "Letter", because of the excessive BDD size. The LRP for this benchmark has a lower node count and can be easily computed.

The experimental results show that LRPs can be used as an alternative representation for multi-valued functions. The efficient manipulation algorithms for LRPs that can be used for decomposition are presented in [32].

Acknowledgements

The work on implicit representations for robotics applications is partially supported by an Intel grant for development of undergraduate robotics laboratories.

Our implementation of implicit algorithms is based on state-of-the-art Binary Decision Diagram package BuDDy, Release 1.7 [34], by Jørn Lind-Nielsen, Department of Information Technology, Technical University of Denmark.

Benchmark	**In#**	**Val#**	**Term#**	**BEMDD**	**T1, c**	**LRP**	**T2, c**
Audiology	69	154	200	2437	0.88	**1123**	0.50
Balance	4	20	625	**85**	0.05	408	0.28
Breastc	9	90	699	3490	0.33	**2116**	0.44
Bridges1	9	29	108	**304**	0.05	370	0.43
Bridges2	10	32	108	**354**	0.06	400	0.05
Car	6	21	1728	**52**	0.33	552	0.66
Chess1	6	40	28056	**6367**	6.10	17939	17.19
Chess2	36	73	3196	**4210**	4.12	6498	5.27
Cloud	6	48	108	451	0.01	**422**	0.05
Employ2	7	29	18000	**51**	3.41	1980	10.60
Flag	28	133	194	4887	0.44	**1499**	0.28
Flare1	10	33	969	**298**	0.28	704	0.44
Letter	16	256	20000	-	300.0	**82743**	37.62
Monks1tr	6	17	124	**118**	0.05	239	0.05
Monks2tr	6	17	169	**138**	0.06	317	0.05
Monks3tr	6	17	122	**119**	0.06	242	0.05
Mushroom	22	117	8124	**894**	6.32	16861	12.85
Nursery	8	27	12960	**63**	3.24	1049	7.58
Post-oper	8	23	90	**161**	0.06	260	0.06
Programm	12	42	20000	**16330**	8.56	51708	23.68
Sensory	11	36	576	1111	0.22	**696**	0.33
Sleep	9	83	62	697	0.06	**429**	0.06
Tic-tac-toe	9	27	958	**536**	0.27	1453	0.49
Trains	32	105	10	235	0.06	**173**	0.06
Zoo	16	39	101	**216**	0.06	300	0.11
alet	18	180	1410	18968	2.20	**6332**	1.65
c3a	14	46	10	160	0.01	**94**	0.01
c3b	14	48	10	197	0.01	**95**	0.01
c6a	13	61	10	247	0.01	**112**	0.05
c6b	13	66	10	239	0.01	**115**	0.06
d4	14	55	20	402	0.01	**169**	0.06
d6	13	87	20	552	0.06	**215**	0.06
d8	32	166	20	1271	0.05	**350**	0.05
Average	12.6	60.2	3600	**1989**	0.98	3491	**0.50**

Table 1. Experimental results using POLO benchmarks.

Explanation of notations used in the table:

In# – the number of inputs
Val# – the sum total of the input values
Term# – the number of cubes in the input file
BEMDD – the number of nodes in the BEMDD after variable reordering by sifting algorithm
T1 – the time needed to build the BEMDD
LRP – the number of nodes in the LRP after variable reordering by sifting algorithm
T2 – the time needed to build the LRP

References

1 C.Y.Lee. “Representation of switching circuits by binary-decision programs”. *Bell Syst. Tech. J.*, Vol. 38, pp. 985-999, July 1959.

2 S.B.Ackers, “Binary Decision Diagrams”. *IEEE Trans. on Computers*, Vol. C-27, pp. 509-516, June 1978.

3 R. E. Bryant. "Graph-Based Algorithms for Boolean Function Manipulation", *IEEE Trans. on Comp.*, Vol. C-35, No. 8 (August, 1986), pp. 677-691. Reprinted in M. Yoeli, *Formal Verification of Hardware Design*, IEEE Computer Society Press, 1990, pp. 253-267.

4 C. Yang, V.Singhal. M.Ciesielski. “BDD Decomposition for Efficient Logic Synthesis”. *Proc. of IWLS’99*.

5 T.Kam, T.Villa, R.Brayton, A.Sangiovanni-Vincentelli. *Synthesis of Finite State Machines: Functional Optimization*. Kluwer Academic Publishers, 1997.

6 R.E.Bryant. “Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams”, *ACM Computing Surveys*, Vol. 24, No. 3 (Sept 1992), pp. 293-318. Available at http://www-cgi.cs.cmu.edu/afs/cs.cmu.edu/user/bryant/www/home.html

7 H.R.Andersen. “Introduction to BDDs”. Lecture notes, 1997. http://www.itu.dk/people/hra/notes-index.html

8 Y.T.Lai, M.Pedram, S.B.Vrudhula. “BDD based decomposition of logic functions with application to FPGA synthesis”. *Proc. of DAC ’93*, pp. 642-647.

9 E.Cerny, M.A.Marin. “An approach to unified methodology of combinational switching circuits”. *IEEE Trans. on Computers* C-26, 8 (August 1977), pp. 745-756.

10 *Coudert O., Madre J. C.* “Implicit and Incremental Computation of Primes and Essential Primes of Boolean Functions”. *Proc. of DAC '92*, pp. 36-39. Available at http://www.synopsys.com/news/pubs/research/index.html

11 O.Coudert, J.C.Madre, H.Fraisse, H.Touati. “Implicit Prime Cover Computation: An Overview”. Proc. of SASIMI (Synthesis And Simulation Meeting and Int'l Interchange), Nara, Japan, 1993.

12 O. Coudert, "Two-Level Logic Minimization: An Overview", *Integration*. Vol. 17, No. 2, pp. 97-140, October 1994.

13 R.K.Brayton, G.D.Hachtel, C.T.McMullen, A.L.Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, Dordrecht, 1984.

14 H.-Y. Wang, R.K.Brayton. “Input Don’t Care Sequences in FSM Networks”. *Proc. of ICCAD ’93*, pp. 321-328.

15 H.J.Touati, H.Savoj, B.Lin, R.K. Brayton, A.Sangiovannni-Vincentelli. "State Enumeration of Finite State Machines using BDDs". *Proc. of ICCAD '90*, pp. 130-133.

16 R.K.Ranjan, A.Aziz, R.K. Brayton, B.Plessier, C.Pixley. "Efficient BDD Algorithms for FSM Synthesis and Verification". *Proc. of IWLS’95*, Lake Tahoe.

17 B.Lin. A.R.Newton. “Implicit Manipulation of Equivalence Classes Using BDDs”. Proc. of ICCAD 1991, pp. 81-85.

18 T.Kam, T.Villa, R.Brayton, A.Sangiovanni-Vincentelli. “A Fully Implicit Algorithm for Exact State Minimization”. *Proc. of DAC ’94*, pp.684-690.

19 T.Kam, T.Villa, R.Brayton, A.Sangiovanni-Vincentelli. *Synthesis of Finite State Machines: Functional Optimization.* Kluwer Academic Publishers, 1997.

20 T.Villa, T.Kam, R.Brayton, A.Sangiovanni-Vincentelli. *Synthesis of Finite State Machines: Logic Optimization.* Kluwer Academic Publishers, 1997.

21 M.Damiani. “The State Reduction of Nondeterministic Finite-State Machines”. *IEEE Trans. on CAD*, Vol. 16, No. 11, Nov 1997, pp. 1278-1291.

22 H.-Y. Wang, R.K.Brayton. “Permissible Observability Relations in FSM Networks”. *Proc. of DAC ’94*, pp. 677-683.

23 H.Cho, G.D.Hatchel, F.Somenzi. “Redundancy Indentification/ Removal and Test Generation for Sequential Circuits Using Implicit State Enumeration”. *IEEE Trans. on CAD*. Vol. 12, No.7. July 1993, pp. 935-945.

24 J.Hartmanis, R.E.Stearns. *Algebraic Structure Theory of Sequential Machines.* Prentice Hall, 1966.

25 L.Jozwiak, “Information Relationships and Measures: An Analysis Apparatus for Efficient Information System Synthesis”, *Proc. of 23d EUROMICRO*, Budapest, Hungary, Sept 1997, pp. 13-23.

26 A.Mishchenko, C.Files, M.Perkowski. “Implicit Algorithms for Multi-Valued Input Support Manipulation”. *Proc. of 4th Intl. Workshop on Boolean Problems*, September 2000, Freiberg, Germany.

27 T. Luba, R.Lasocki. “Decomposition of multiple-valued Boolean functions”. *Applied Math and Computer Science*, 4(1): 125-138, 1994.

28 Y.T.Lai, M. Pedram. “EVBDD-based Algorithms for Integer Programming, Spectral Transformations and Functional Decomposition”. *IEEE Trans. on CAD*, Vol. 13, No. 8, August 1994, pp. 959-975.

29 Y.T.Lai, K.R.R.Pan, M. Pedram. “OBDD-Based Functional Decomposition: Algorithm and Implementation”. *IEEE Trans. on CAD*, Vol. 15, No 8, August 1996, pp.977-990.

30 C. Files, M.A.Perkowski. “New Multi-Valued Functional Decomposition Algorithms Based on MDDs”. Accepted to *IEEE Trans. on CAD*, 2000.

31 S.Grygiel, M.Perkowski, M.Marek-Sadowska, T.Luba, L.Jozwiak. “Cube Diagram Bundles: A New Representation of Strongly Unspecified Multi-Valued Functions and Relations”. *Proc. of ISMVL’97*, Halifax, Nova Scotia, Canada, May 1997, pp. 287-292.

32 S. Grygiel, M.Perkowski. “A New General Purpose Representation for Multiple-Valued Functions and Relations”. *Unpublished manuscript*, 1999.

33 http://www.ee.pdx.edu/polo/

34 J. Lind-Nielsen. Binary Decision Diagram Package BuDDy, v.1.7, Nov 1999, http://www.itu.dk/research/buddy/index.html