Functional Decomposition of MVL Functions using Multi-Valued Decision Diagrams

Craig Files[1]   Rolf Drechsler[2]   Marek A. Perkowski[1]

[1]Department of Electrical Engineering
Portland State University
Portland, OR 97207-0751, USA
email: cfiles@ee.pdx.edu

[2]Institute of Computer Science
Albert-Ludwigs-University
79110 Freiburg im Breisgau, Germany
email: drechsle@informatik.uni-freiburg.de

Abstract

In this paper, the minimization of incompletely specified multi-valued functions using functional decomposition is discussed. From the aspect of machine learning, learning samples can be implemented as minterms in multi-valued logic. The representation, can then be decomposed into smaller blocks, resulting in a reduced problem complexity. This gives induced descriptions through structuring, or feature extraction, of a learning problem. Our approach to the decomposition is based on expressing a multi-valued function (learning problem) in terms of a Multi-valued Decision Diagram that allows the use of Don't Cares. The inclusion of Don't Cares is the emphasis for this paper since multi-valued benchmarks are characterized as having many Don't Cares.

1. Introduction

This paper explores functional decomposition as it is extended to the synthesis of *Multi-Valued Logic Networks* (MVLNs) and to the concept of *Machine Learning* (ML). The two problems are disjoint, given that many synthesis problems for MVLNs are completely specified or nearly completely specified functions. While functions in ML tend to have 99.9% *Don't Cares* (DCs) in their respective learning problems.

Decomposition was proposed by Ashenhurst in the 1950's [2] as a method of Boolean multi-level logic minimization. While this approach has been known for many years, it wasn't until recently that this approach could be utilized because of the large computation procedures that are required. Lately, though, while much research has focused on decomposition as applied to FPGA design synthesis, several researchers have shown general decomposition

methods by expressing Boolean functions as *Binary Decision Diagrams* (BDDs) [7, 12].

The approach proposed in this paper is to decompose MVL functions using *Multi-valued Decision Diagrams* (MDDs) [10] as the underlying data structure. It was shown that MDDs have a direct correspondence to the problem considered (analogously to BDDs in the Boolean domain). We consider, especially, the aspect of using DCs in the minimization process and outline the occurring difficulties. In contrast to previous papers on MVL decomposition [9, 11] we consider functions with MV outputs.

The concept of using decomposition in ML is to reduce a given function specified by a set of care minterms (samples) to a composition of smaller functions (attributes in ML terminology). The result is a set of expressions that describe suitable *intermediate concepts*. Each of these *intermediate concepts* can then be further decomposed, leading to expressions that form a more comprehensible description of the learned concepts. The advantage of using decomposition to obtain useful *intermediate concepts* is that it leads to a result being in a hierarchy of compositions that could be illustrated as a tree structure. This tree structure gives the original function a hierarchy of sub-functions and variables, which leads to learning that is faster, involves smaller error and gives better explanation of the learned concepts.

In terms of decomposing functions that are (nearly) completely specified, the decomposition process involved may produce functions with DCs. In [8], the authors show that functions can be minimized by using generalized DC values to assume any value in the range of the input function (other than the previously used Boolean values). Thus, the decomposition algorithms proposed in the paper can be easily applied to MVLNs. Recently, more and more interest has been in the design of such networks. Because of this interest, several design methods were proposed to reduce the final design and size of the network. In these cases, though,

most of them were only of theoretical nature or have not been applied to large examples.

Experimental results are given to show the efficiency of our approach. We use a benchmark set from the area of machine learning that is characterized by the fact that the functions are "real" MVL functions and that they contain many DCs.

The paper is structured as follows: In Section 2 preliminaries are given, i.e., notations and definitions are introduced. The method of decomposition is described in Section 3. Section 4 addresses the problem of the implementation and the choice of the underlying data structure of our algorithm. In Section 5 experimental results are described. Finally the results are summarized.

2. Preliminaries

This section provides the notations which are the basics of multi-valued logic and are important for the understanding of this paper.

2.1. Multi-valued Decision Diagrams

It is well known that each Boolean function $f : \mathbf{B}^n \to \mathbf{B}$ can be represented by an ordered BDD [5], i.e., a directed acyclic graph where a Shannon decomposition is carried out in each node.

Obviously, BDDs can be extended to represent functions $f : \mathbf{B}^n \to \{0, .., k-1\}$ and the resulting graphs are denoted as *Multi-Terminal BDDs* (MTBDDs). The operations on MTBDDs can be carried out as efficiently as in the case of two terminals [6].

It is straightforward to extend MTBDDs to MDDs [10] representing functions $f : \{0, .., k-1\}^n \to \{0, .., k-1\}$. For this, each internal node has k outgoing edges[1]. In [10], it was shown that the efficient operations known for BDDs can also be carried out on MDDs using a *case*-operator (multi-valued Shannon decomposition operator).

The extension or addition of *Don't Cares* (DC) to an MDD is fairly straightforward by representing the function as $f : \{0, .., k-1\}^n \to \{0, .., k\}$. Thus, each internal node still has k outgoing edges, but has the ability of representing $k+1$ values. The $(k+1)$-th value is used to represent a DC value in the function.

A DD is called *ordered* if each variable is encountered at most once on each path from the root to a terminal and if the variables are encountered in the same order on all such paths. A DD is called *reduced* if it does not contain vertices either with isomorphic sub-graphs or with all successors pointing to the same node.

In this paper, only reduced, ordered MDDs are considered.

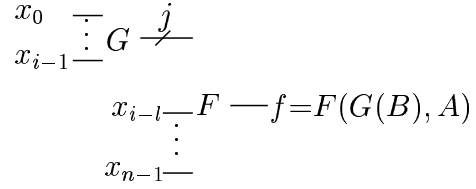[1] In our application all variables are defined over the same set of values.

Figure 1. Curtis Decomposition

3. Functional Decomposition

In this section, the basic principles of the decomposition of binary and MVL functions are described. For simplicity of the presentation most of the examples are in the binary case, but can easily be related directly to MVL cases. Finally, the influence of DCs used in the decomposition process is discussed.

3.1. Generalized Functional Decomposition

Definition 1 A function $f(x_0, x_1, \ldots, x_{n-1})$ is decomposable under *bound set* $\{x_0, \ldots, x_{i-1}\}$ and *free set* $\{x_{i-l}, \ldots, x_{n-1}\}$, $0 < i < n, 0 \le l$ if and only if f can be represented as the composite function $F(G_0(x_0, \ldots, x_{i-1}), \ldots, G_{j-1}(x_0, \ldots, x_{i-1}), x_{i-l}, \ldots, x_{n-1})$, where $0 < j < i-l$. If l equals 0 then f is said to be *disjunctively decomposable*, otherwise, it is known as *non-disjunctively decomposable* [3, 4].

The principle idea of the decomposition using the notations from Definition 1 is shown in Figure 1. Note, that because of the complexity of *non-disjunctive decompositions*, *disjunctive decomposition* is the method used in this paper.

Definition 2 Given a k-valued, completely specified function f, with a bound set B, and free set A, then for the partition $A|B$, a *partition matrix representation of f* is defined as a rectangular array, where the columns correspond to the variables in the bound set, and the rows correspond to the variables in the free set.

Using these definitions the following is found:

1. The array has $k^{\{B\}}$ columns and $k^{\{A\}}$ rows.

2. Given a k-valued function f, with a bound set B, then the corresponding partition matrix has l distinct columns, where l is known as the *Column Multiplicity* (CM) of a partition.

3. The CM for the function can be reduced if the function is incompletely specified, by finding columns
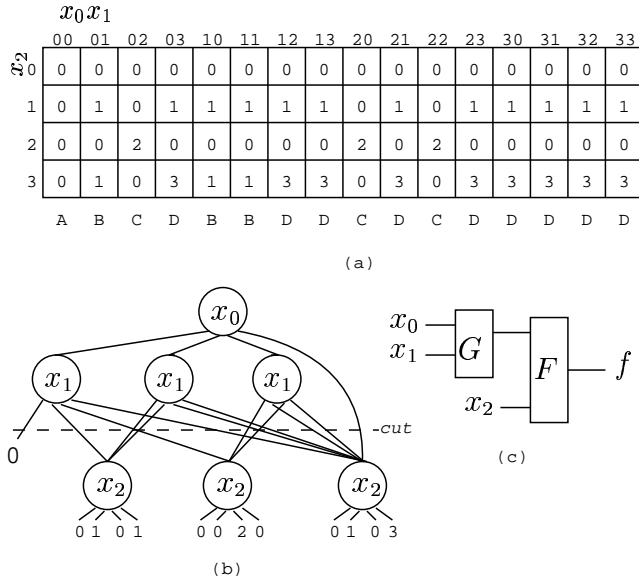
x_0x_1	00	01	02	03	10	11	12	13	20	21	22	23	30	31	32	33
x_2 0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	0	1	1	1	1	1	0	1	0	1	1	1	1	1
2	0	0	2	0	0	0	0	0	2	0	2	0	0	0	0	0
3	0	1	0	3	1	1	3	3	0	3	0	3	3	3	3	3
	A	B	C	D	B	B	D	D	C	D	C	D	D	D	D	D

(a)

(b)

(c)

Figure 2. Example for MVL decomposition

that are *compatible* and combining the two columns by setting don't care values. By compatible, for every row, the possible output value-sets of the first column (a number or a DC) intersect the non-empty sets of the corresponding output value-sets of the second column.

4. Thus, to represent f as a composite function in the form

$$F(G_0(), \ldots, G_{j-1}(), x_i, \ldots, x_{n-1})$$

where each G function has inputs $(x_0, \ldots, x_{i-1})$ then $j = \lceil log_k l \rceil$, G functions are needed.

Example 1 Consider the 4-valued function in Figure 2, where the partition of a given function is shown in Figure 2(a). Notice that the number of distinct columns is four (labeled as A, B, C and D), the function can be represented as a composite function with $\lceil lg3 \rceil$=2, G functions. The resulting decomposition is denoted as $F(G_0(x_0, x_1), x_2)$.

3.2. BDD/MDD Based Decomposition

In [7], a method for detecting decompositions based on the concept of a *cut_set* in a BDD representation of Boolean functions was presented. (In the following we briefly review the main idea as applied to MDDs; for more BDD details see [7].)

Definition 3 Given a bound set $\{x_0, \ldots, x_{i-1}\}$ and free set $\{x_i, \ldots, x_{n-1}\}$ for a function f, then if a DD is constructed with variable ordering $\{x_0, \ldots, x_{i-1}\} < \{x_i, \ldots, x_{n-1}\}$ the *cut_set*$(i-1)$ is the number of distinct columns (*column multiplicity*) for the given bound and free sets. The *cut_level* is defined as the split between the two sets.

Example 2 Given the multi-valued function from Example 1, the corresponding MDD is created by placing the *bound* variables $\{x_0, x_1\}$ on the top of the BDD, while the *free* variables $\{x_2\}$ are on the bottom. Note, that the order of the variables within the *bound* set or the *free* set has no effect on the partition found. The MDD obtained is shown in Figure 2(b). Notice that the number of nodes below the *cut_level*, i.e., the number of x_2 nodes plus the number of terminal nodes having a pointer cross the *cut_level*, is equal to four. The resulting decomposition is shown in Figure 2(c).

3.3. Don't Cares

One of the biggest advantages of functional decomposition is that after a function is broken up into smaller blocks, DCs are (possibly) introduced into these smaller blocks. These DCs can then be used for optimization in the next level of decomposition,i.e., decomposing either the resulting G functions or the F function.

The following example explores the ideas of DCs in Boolean functions, but note that the idea can be directly applied to MVL functions. (A Boolean function and the use of partitions are for simplicity and ease of explanation.)

Example 3 Given the Boolean function shown in Figure 3(a) and resulting partition in Figure 3(b), the function can be decomposed into $\lceil lg3 \rceil$=2, G functions. Using an encoding scheme of A={0,0}, B={0,1}, and C={1,1}, G_0=0 0 0 1 or G_0=x_0 AND x_1, while G_1=0 1 1 1 or G_1=x_0 OR x_1. The resulting partition matrix is shown in Figure 3(c), where 4 DCs are introduced into the function. The DCs come from the above encoding, where the case $\{1,0\}$ does not exist. Now, by swapping the variables G_0 and x_3 the partition matrix in Figure 3(d) is found. The partition $\{x_3G_1\}|\{x_2G_0\}$ has CM of 2, while $\{x_2G_0\}|\{x_3G_1\}$ also has CM of 2. Notice that without the addition of DCs (say the values were forced to 0 instead) then the resulting CM would be larger than 2.

The example shows the importance of introducing DCs into the partition matrix. Notice that encoding is also a big factor in the final outcome of the decomposition. The concept of encoding, though, has not been evaluated in previous approaches, where a single output line is constructed with a multi-valued output to carry all possible data[13]. The problem with not using the encoding scheme is that it may lead to functions that do not decompose. The encoding,
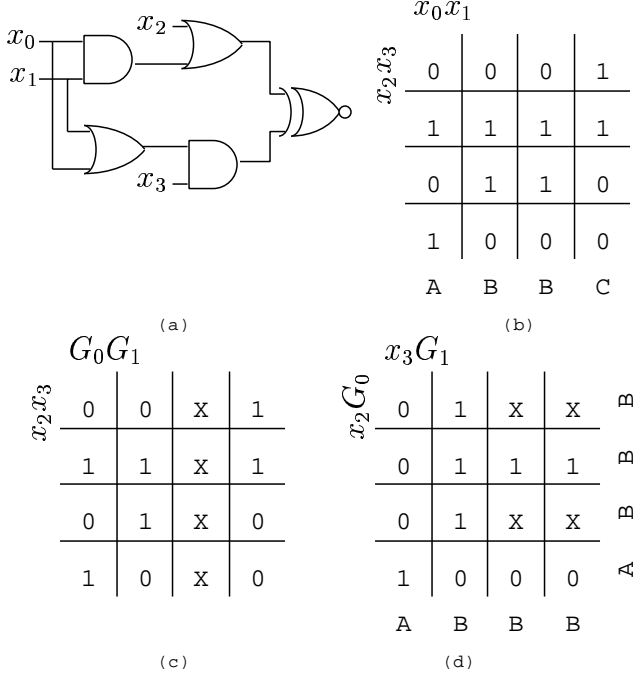
Figure 3. Example for using Don't Cares

$x_0 x_1$ (columns), $x_2 x_3$ (rows) — (b)

A	B	B	C
0	0	0	1
1	1	1	1
0	1	1	0
1	0	0	0

$G_0 G_1$, $x_2 x_3$ (rows) — (c)

0	0	X	1
1	1	X	1
0	1	X	0
1	0	X	0

$x_3 G_1$, $x_2 G_0$ (rows) — (d)

A	B	B	B	
0	1	X	X	B
0	1	1	1	B
0	1	X	X	B
1	0	0	0	A

used in this paper, is done by using k-valued logic symbols, where k is given by the original input function.

A problem arises, though, when functions become incompletely specified, since the detection of CM becomes very difficult. From Example 2, the CM was calculated using by counting the number of "columns" at a certain *cut_set* level. In determining CM in the presence of DCs, each node below the *cut_level* may represent an incompletely specified function. The CM is then determined by computing the possible compatibility between each of these nodes and then using some search strategy, find the minimal CM. Possible approaches include: reducing the problem to set covering, graph coloring, or heuristic methods[13].

In the case of MVL, the problem of determining a minimal CM (often) becomes reduced because the possibility of having *compatible* nodes decreases with an increase in the number of values represented in the function. But, in the case of machine learning problems, in general, the functions are highly unspecified, e.g., for large functions, 99.9% of the function may be unspecified. Also, by definition, given a CM for a partition $\lceil log_k CM \rceil$ G functions must be used in the composite function. As the size of k grows, finding the exact minimal CM becomes less of an issue because the idea of reducing CM is to reduce the number of G functions.

Example 4 Given a partition, using 5-valued logic, the CM is originally found to be 11 (before setting DC val-

ues). To represent this decomposition two G functions ($\lceil log_5 11 \rceil = 2$) must be used. The only way that the number of G functions will be reduced is if 7 columns can be combined. By analyzing the partition, it might be obvious that this is an impossible task. (Even if it is possible, many DC values must be set to literal values.) By not combining the DC columns the current partition will result in a decomposition with 5^2-11=14 columns with DCs. Thus, the function F, found from $f=F(G(A), B)$, will have an additional $14 * 5^{\{B\}}$ DCs. Notice, though, that encoding now becomes a problem because there are $\begin{pmatrix} 25 \\ 11 \end{pmatrix}$=4,457,400 possible encodings.

4. Implementation

The main goal in logic synthesis is finding a "good" result through some figure of merit. In this paper, the result of decomposing a MVL function must be general or robust, in that, the use of the realization technology for the decomposed function is unknown. Also, when using the concept of functional decomposition in machine learning, the main concern is determining the "pattern" of the object. Thus, it is unnecessary to have different figures of merit for each technology as long as a "good" figure of merit is chosen to represent the generality of a pattern.

Thus, an appropriate method for representations that do not require distinction between different gate types is called *Decomposed Function Cardinality* (DFC). This idea was first proposed by Abu-Mostafa in 1988 [1] for binary functions and is extended here for MV logic.

Definition 4 A function $f(x_0, x_1, \ldots, x_{n-1})$ with k valued logic and m outputs, has *cardinality* $k^n * m$. DFC is defined as the sum of the function cardinality of each decomposed partial block. A non-decomposable function is defined as a function that has a DFC$> k^n * m$.

It is a "good" representation of the complexity or "patterness" of a function, since a function is a set of ordered pairs and, as with any set, has a definite property in its number of elements or cardinality. Thus, for a function that has multiple decompositions, DFC may be used to find the minimum combined component cardinality of all the decompositions of that function. Because of the simplistic representation of DFC, it is also a "good" figure of merit for evaluating function implementations in most of the current technologies.

Example 5 Given the multi-level function $f=F(G_0(x_0, x_1), G_1(x_2, x_3), x_4, x_5)$, where f has six primary inputs, is 4-value, and has 1 output. Then the cardinality of f is $4^6 * 1$=4096 and the DFC of the given multi-leveled decomposition is $4^2 + 4^2 + 4^{(1+1+2)} = 288$.

For a function that has multiple decompositions, DFC may be used to find the minimum combined component cardinality of all the decompositions of that function. The objective of decomposition (in this paper) is to find the decomposition that produces the smallest DFC. Notice, though, that finding the exact or smallest DFC is an NP-complete problem, i.e., evaluating all partitions (discounting the NP-complete problems involved with CM and encoding).

Column compatibility, in this paper, is found by a very simple and quick heuristic that attempts at matching columns in a maximal clique. Encoding is done in a sequential manner, i.e., the column labeled A is given the value 0; B=1; C=2, etc.

The heuristic for selecting partitions, evaluates all possible partitions, the partition that results in the smallest DFC, of a simple decomposition (first level decomposition), is used in the next level decomposition. Note, that the smallest level DFC, doesn't always result in the smallest overall DFC.

Even if this is still a heuristic, it is not very fast, in that, it must evaluate on the order of 2^n (n-input variables) partitions, which implies calling a dynamic swap procedure in the MDD on the order of 2^n times. The reason for not implementing "faster" heuristics is because one of the best attributes in MVL functions is that it has a reduced number of input variables, i.e., fewer partitions to evaluate. For example, given a 10-input, 10-valued, 1-output function, there are 2^{10} partitions to evaluate, but if the function was encoded to a Boolean function then there would be 30-input variables, implying 2^{30} partitions to evaluate. (The solution may also be constrained to groups of binary variables that come from encoding the same MV symbol so that they are always taken all together to bound or free sets.)

The recursive procedure's pseudo-code of the algorithm is given in Figure 4. The first statement checks to see if the number of variables is less than three, if it is, then this path of the decomposition is completed. The program then evaluates all the possible (*disjunctive*) decompositions for the given function. The partition that has the smallest (single level) DFC is chosen as the best. This partition is then used to break up the function f into its components F and G_0 through G_{n-1}. Each of these new functions is then decomposed. The resulting DFC returned from each decomposition process is added up and compared to the cardinality of the function. If the DFC found is larger than the cardinality of the original function then the function is said to be *undecomposable* (at whatever level of decomposition).

5. Experimental Results

In this section, the experimental results of our decomposer, FREDMVL, while running on a *HP 700* workstation are presented. All run times are given in CPU minutes.

```
int decompose(function f) {
  if (Num_vars<3)  then
    return cardinality of f;
  evaluate all possible partitions;
  // given partition with smallest DFC
  // create F and G0 through Gn-1
  DFC=decompose(F);
  for i=0 to (n-1)
    DFC=DFC+decompose(G(i));
  // if function is non-decomposable
  // return cardinality of the f
  if (DFC < cardinality of f) return DFC;
  else  return cardinality of f;
}
```

Figure 4. Algorithm

name	in	out	k	card.	DFC	time
balance	4	1	5	625	375	0.1
breastc	9	1	10	10^9	7025	207
flare1_1	10	1	7	7^{10}	1797	7
flare1_2	10	1	7	7^{10}	3512	8
flare1_3	10	1	7	7^{10}	1751	8
flare2_1	10	1	7	7^{10}	1413	7
flare2_2	10	1	7	7^{10}	2867	9
flare2_3	10	1	7	7^{10}	1751	8
hayes	4	1	5	625	314	0.1

Table 1. Benchmark characteristics

The benchmarks are taken from the *UCI repository of Machine Learning*[2]. Some characteristic information about the benchmarks is given in Table 1. The files labeled *flare1_** and *flare2_** come from the same output functions of *flare1* or *flare2*, respectively.

The run times mainly stem from the size of the initial networks, as in the case of each of the *flare* benchmarks, 2^{10} partitions are evaluated for the first level decomposition alone. The reason for the delay in the *breastc* benchmark is because of the large number of DCs in the initial network. Thus, checking for CM is a very difficult task, as there are many partitions with over 300 columns, that by combining columns, the CM drops to less than 20. In fact, the partition that had the smallest DFC in the first level decomposition originally had a CM of 345 which was reduced to a CM of 7 by setting DC values.

6. Conclusions

We presented a new approach to functional decomposition of MVL functions. Our algorithm makes use of MDDs as the underlying data structure and by this becomes applicable to larger problem instances. We considered the problem of multi-level minimization with special emphasis on using DCs. The quality of our algorithm has been demonstrated by application to a set of benchmarks from the area of machine learning.

Finally, we want to point out one problem that has to be faced in our algorithm: The number of nodes in a level of the MDD might become large. Since our algorithm uses operations that are linear in the number of nodes per level the run times may largely increase. In [10], it was reported that by encoding MDDs on BDDs the runtime of an MDD package can be much improved. It is the focus of current work, whether an MDD or a BDD based data structure should be preferred in decomposing MVL functions.

References

[1] Y.S. Abu-Mostafa. *Complexity in Information Theory*. Springer-Verlag, New York, 1988.

[2] R.L. Ashenhurst. The decomposition of switching functions. In *Int'l Symp. on Theory Switching Funct.*, pages 74–116, 1959.

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

[4] J.P. Roth, R.M. Karp. Minimization over Boolean graphs. In *IBM Journal of Research and Development*, pages 227–38, 1962.

[5] R.E. Bryant. Graph - based algorithms for Boolean function manipulation. *IEEE Trans. on Comp.*, 8:677–691, 1986.

[6] E. Clarke, M. Fujita, P. McGeer, K. McMillan, J. Yang, and X. Zhao. Multi terminal binary decision diagrams: An effcent data structure for matrix representation. In *Int'l Workshop on Logic Synth.*, pages P6a:1–15, 1993.

[7] Y.-T. Lai, M. Pedram, and S.B.K. Vrudhula. BDD based decomposition of logic functions with application to FPGA synthesis. In *Design Automation Conf.*, pages 642–647, 1993.

[8] L. Lavagno, S. Malik, , R.K. Brayton, and A.L. Sangiovanni-Vincentelli. MIS-MV: optimization of multi-level logic with multiple-valued inputs. In *Int'l Conf. on CAD*, pages 560–563, 1990.

[9] T. Luba. Decomposition of multiple-valued functions. In *Int'l Symp. on multi-valued Logic*, pages 256–261, 1995.

[10] A. Srinivasan, T. Kam, S. Malik, and R.E. Brayton. Algorithms for discrete function manipulation. In *Int'l Conf. on CAD*, pages 92–95, 1990.

[11] Y. Watanabe and R.K. Brayton. Heuristic minimization of multiple-valued relations. *IEEE Trans. on CAD*, 12, 1995.

[12] C. Scholl and P. Molitor. Efficient ROBDD based computation of common decomposition functions of multioutput boolean functions. In G. Saucier and A. Mignotte, editors, *Novel Approaches in Logic and Architecture Synthesis*, pages 57–63. Chapman & Hall, 1995.

[13] M. Perkowski, M. Marek-Sadowska, L. Jozwiak, T. Luba, S. Grygiel, M. Nowicka, R. Malvi, Z. Wang, and J.S. Zhang. Decomposition of multiple-valued relations In *International Symposium on Multi-Valued Logic*, May 1997.