# Implicit Algorithms for Multi-Valued Input Support Minimization

**Alan Mishchenko, Craig Files, Marek Perkowski**
Portland State University
Department of Electrical and Computer Engneering
Portland, OR 97207, USA
[alanmi,cfiles,mperkows]@ee.pdx.edu

**Bernd Steinbach, Christina Dorotska**
Freiberg University of Mining and Technology
Institute of Computer Science
D-09596 Freiberg, Germany
[steinb,dorotsk]@informatik.tu-freiberg.de

**Abstract**

*We present an implicit approach to solve problems arising in decomposition of incompletely specified multi-valued functions and relations. We introduce a new representation based on binary-encoded multi-valued decision diagrams (BEMDDs). This representation shares desirable properties of MDDs, in particular, compactness, and is applicable to weakly-specified relations with a large number of output values. This makes our decomposition approach particularly useful for data mining and machine learning.*

*Using BEMDDs to represent multi-valued relations we have developed two complementary input support minimization algorithms. The first algorithm is efficient when the resulting support contains almost all initial variables; the second is efficient when it contains only a few. We propose a simple heuristic measure to choose which algorithm to use. Numerical results over a set of multi-valued benchmarks provide experimental evidence for the efficiency of our approach.*

## 1 Introduction

Over the years, a number of researchers have been working on multi-valued decomposition as a vehicle for multi-valued logic design, machine learning, and data mining. Historically, explicit methods based on multi-valued cubes and cube partition algebras appeared first and are still used [1]. Following the emergence of BDDs, new approaches have been created that exploit the structure of decision diagrams in order to solve decomposition problems [2].

Explicit methods have serious performance drawbacks. As a result, only simple benchmarks can be solved in a reasonable time using Boolean or multi-valued cubes. BDD- and MDD-based approaches, on the other hand, can be applied to much larger benchmarks. Their applicability, however, is limited because the known decomposition algorithms are not formulated to exploit the potential of this representation.

Since early 1990's, we have developed a number of approaches to multi-valued decomposition, in particular, decomposition based on labeled rough partitions [3] and decomposition based on classical MDDs [4].

Our recent research has been directed towards 1) solving problems, for which efficient MDD-based algorithms are not known, 2) overcoming limitations inherent in MDD-based representation, and 3) improving the performance of known MDD-based procedures.

As a result of these efforts, we came up with a modified representation, called binary-encoded MDDs (BEMDDs), which shares the advantages of MDDs (compactness and efficient manipulation) while overcoming some of their limitations. In particular, in order to represent a multi-valued relation with many output values using MDDs, it is necessary to create as many terminal nodes as there are subsets of output values. For example, in the case of a 16-valued relation, there are $2^{16}$-1 non-empty combinations of output values, each of which should be represented by a separate node in the MDD.

BEMDDs, on the other hand, provide an easy way to encode output values using the minimum number of binary variables. Additionally, BEMDDs exploit efficient machinery of specialized BDD operators, which recently received attention due to the success in BDD-based formal verification algorithms. In particular, BDD operators such as quantification, variable replacement, and composition can be used to improve the performance of multi-valued decomposition.

While exploring the potential of BEMDDs, we implemented and significantly improved the performance of the known algorithms for bound set and column multiplicity computation, which constitute the basis of iterative Curtis decomposition. These, as well as decomposition for large bound sets and implicit methods for multi-valued bi-decomposition, are the subject of our forthcoming papers.

In this paper, we present the fundamentals of BEMDDs and their use in solving the problem of input support minimization arising in multi-valued weakly specified functions and relations.

To prevent words “functions and relations” from appearing too often in this paper, we will use the term MISFs meaning “Multi-valued-input multi-valued-output Incompletely Specified Functions”. Unless specifically noted, the presented material is true for multi-valued incompletely specified relations.

The rest of the paper is organized as follows. **Section 2** describes the representation of MISFs using BEMDDs. **Section 3** introduces the basic concepts of input support minimization. **Section 4** presents the general strategy of support minimization and two possible branch-and-bound algorithms to determine whether a multi-valued variable can be added to (removed from) the support set. These algorithms rely on efficient BEMDD traversal routine as described in **Section 5**. Next, in **Section 6**, we derive an approximate measure, which can be used to determine which of the two presented support minimization techniques works better for a given MISF. Experimental results are given in **Section 7** and the paper is concluded in **Section 8**.

## 2 Representation of MISFs

As stated in the introduction, we advocate the use of BEMDDs as the representation of choice for MISFs. The motivation is that BEMDDs are efficient for large functions and lead to improved decomposition procedures compared to other known representations: multi-valued cubes and cube partitions [1], edge-valued BDDs [2], and classical MDDs [4].

In our approach, multi-valued variables are encoded using the smallest possible sets of binary variables. Thus, a k-valued variable requires at least $\lceil \log2(k) \rceil$ 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: $\{ \bar{a}_1\bar{a}_2\bar{a}_3,\ \bar{a}_1\bar{a}_2a_3,\ \bar{a}_1a_2\bar{a}_3, \bar{a}_1a_2a_3, a_1\bar{a}_2\bar{a}_3 \}$. (Here the prime symbol following a literal means negation.)

If k is not an integer power of two, the minimum length binary encoding results in $2^{\lceil \log2(k) \rceil} - k$ 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 the input variables is limited to only those minterms that provide codes for the values of multi-valued variables. This, however, increases the BEMDD size and makes traversal routines complicated.

Therefore, 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. 1.

| | 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 | 10 | 6 |
| 10 | 1 | 3 | 9 | 5 |

Fig. 1. Two ways to encode ten values using four binary variables

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

Fig. 2. The truth table for MISF F(A,B)

Currently, as the default encoding of values, we use the encoding shown on the left of Fig. 1. Meanwhile we continue to experiment with other encodings.

Because outputs of most MISFs are multi-valued, we use an encoding scheme to represent the outputs. The variables used for this purpose are called *output-value-encoding variables* (or simply, *value variables*). Traversal algorithms typically require the ordering of the value variables below other variables in the BEMDD.

**Definition**. Let us designate the domain of a multi-valued variable $A_i$ as $D_{Ai}$. An MISF $\Re$ over a set of multi-valued input variables $\{A_i\}$, with multi-valued output Z having domain $D_z$, is a mapping from the Cartesian product of the input variable domains into the domain of subsets of the output values:

$$\Re( A_1,A_2,\ldots,A_n ): D_{A1}\times D_{A2}\times\ldots\times D_{An} \to 2^{Dz}.$$

**Definition**. The set of minterms (vertices) of the combined input domain of $\Re$ such that the output of $\Re$ is the set of all possible output values, is called the *don’t-care set*. The set of all other minterms of the domain of $\Re$ is called the *care set*.

Once the input and output multi-valued variables are encoded, the BEMDD representing a MISF can be constructed as a binary relation, assuming certain ordering of the encoding variables. This relation connects encoded input values with the corresponding encoded output values. For example, shown in Fig. 2 is a three-valued output MISF 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).

Suppose the binary variable a encodes multi-valued variable A, variables $b_1$ and $b_2$ encode B, while variables $v_1$ and $v_2$ encode the output values of F(A,B). If ternary values { 0,1,2 } are encoded as { 00,01,1- }, relation R, which represents F(A,B), is expressed using variables a, $b_1$, $b_2$, $v_1$, and $v_2$ as follows:

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

Each of the six terms in the above formula is obtained directly by encoding a cell of the map in Fig. 2. The Karnaugh map and the BEMDD for R are shown in Figs. 3 and 4.

| $v_1v_2$ \ $ab_1b_2$ | 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. 3. The Karnaugh map for MISF F(A,B)

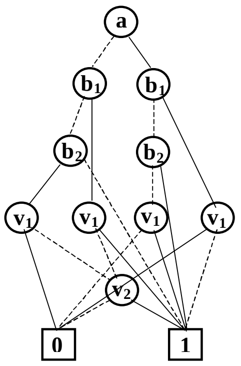

Fig. 4. The BEMDD for MISF F(A,B)

## 3 Input Support Minimization

### 3.1 Overview

The problem of input support minimization arises when MISFs under consideration are weakly specified. In this case, it is often possible to remove some variables from the support without losing information contained in the function. Decomposition of MISFs with the minimized support typically leads to simpler formulas and more meaningful interpretations in data mining or machine learning.

To our knowledge, the problem of input support minimization for MISFs has been addressed by only a few researchers [5,6]. A complete bibliography is given in [5]. The known algorithms are based on explicit representations using cubes and so their use is limited only to functions with small cube covers.

For incompletely specified Boolean functions, an elegant approach to input support minimization has been created [7]. This approach uses bottom-up traversal of BDDs to implicitly enumerate all feasible supports, and then finds the minimum one. Unfortunately, this approach cannot be generalized to the multi-valued case.

Our methods are essentially different from [7] and they favorably compare with the previous results [5,6]. Because we use BEMDDs to represent MISFs, the efficiency of our approach is not limited to problems with small cube covers.

Below two algorithms for multi-valued input support minimization are proposed. The merits of these algorithms are complementary: one of them works well when most of the input variables should remain in the support, while the other works well when only very few variables are needed to represent the function. For intermediary cases, any approach can be used.

### 3.2 Classification of Input Variables

First we restate some definitions, which are given in [8].

Input variables of weakly specified MISFs are divided into three categories: vacuous, inessential, and essential, depending on the amount of information they carry.

**Definition**. A variable is *vacuous* if it does not appear in the MDD (and BEMDD) of the function.

The ternary relation over four binary variables specified using the truth table in Fig. 5 illustrates the two reasons why a variable is vacuous. Variable d is vacuous, because it does not enter any of the cubes and therefore the MDD does not depend on it. Variable c is vacuous because even though it enters the first and the third cube, these cubes can be combined into a single cube without variable c, which means that c does not appear in the MDD as well.

| a | b | c | d | F |
|---|---|---|---|---|
| 0 | 1 | 0 | - | 0 |
| - | 0 | - | - | 1,2 |
| 0 | 1 | 1 | - | 0 |
| 1 | 1 | - | - | - |

Fig. 5. The truth table of F(a,b,c,d) with two vacuous variables (c and d).

| ab\c | 0 | 1 |
|---|---|---|
| 00 | 0 | - |
| 01 | - | 0 |
| 11 | 1 | - |
| 10 | - | 0 |

Fig. 6 a)

| ab | |
|---|---|
| 00 | 0 |
| 01 | 0 |
| 11 | 1 |
| 10 | 0 |

Fig. 6 b)

| ab\c | 0 | 1 |
|---|---|---|
| 00 | 0 | - |
| 01 | - | 0 |
| 11 | 1 | 0 |
| 10 | - | 0 |

Fig. 7)

The truth tables of binary functions used in examples.

**Definition**. A variable is *inessential* if it can be removed from the support of MISF in such a way that no information initially contained in the function is lost, assuming that other variables remain in the support.

**Definition**. A variable is *essential* if it is neither vacuous nor inessential.

Observe that an inessential variable can become essential after removing some variables from the support.

For instance, the binary function in Fig. 6a) depends on three binary variables: a, b, and c. According to the above definition, all these variables are inessential.

Having removed variable c from the support, for example, we get function in Fig. 6b). This function is represents the initial function without the loss of information, because, for every assignment of variables a and b, functions in Fig.6a) and Fig. 6b) assume the same values. Notice that, after removing variable c, variables a and b become essential.

Fig. 7 shows a function that has no inessential variables.

## 3.3 Definition of Support Minimization

In this section, input support minimization based on the concept of information losslessness is defined.

**Definition**. Given the MISF $\Re(x_1,x_2,\ldots,x_k)$ and two assignments of its input variables, $(a_1,a_2,\ldots,a_k)$ and $(b_1,b_2,\ldots,b_k)$, consider sets of the output values of $\Re$ produced by these assignments, $S_1 = \Re(a_1,a_2,\ldots,a_k)$ and $S_2 = \Re(b_1,b_2,\ldots,b_k)$. Sets $S_1$ and $S_2$ are *compatible* iff $S_1 \cap S_2 \neq \varnothing$.

**Definition**. The result of *merging* of two output sets $S_1$ and $S_2$ is output set $S_1 \cap S_2$.

For example, the output sets {1,2} and {0,1} that appear in Fig.2 are compatible and the result of their merging is the output set {1}.

**Definition**. Suppose MISFs $\Re_1$ and $\Re_2$ depend on the same input variables and take the same output values. $\Re_1$ and $\Re_2$ are *compatible* if for each assignment of the input variables, the output sets are compatible. The result of merging $\Re_1$ and $\Re_2$ is MISF $\Re$ satisfying the following condition. For each assignment of the input variables, the set of output values of $\Re$ is the result of merging of the sets of output values of $\Re_1$ and $\Re_2$.

For example, the MISFs over variables a and b represented in the left and right columns of Fig. 6a) can be merged to produce the MISF in Fig. 6b).

**Definition**. MISF $\Re_2(x_{i1},x_{i2},\ldots, x_{im})$, $\{x_{i1},x_{i2},\ldots, x_{im}\} \subseteq \{x_1,x_2,\ldots,x_k\}$, *represents MISF* $\Re_1(x_1,x_2,\ldots,x_k)$ *without the loss of information*, iff (1) for any given assignment of variables $\{x_{i1},x_{i2},\ldots, x_{im}\}$, the outputs of $\Re_1(x_1,x_2,\ldots,x_k)$ corresponding to all possible assignments of other variables in $\{x_1,x_2,\ldots,x_k\}$, are compatible; (2) the output of $\Re_2(x_{i1},x_{i2},\ldots, x_{im})$ is the result of merging of all these compatible outputs of $\Re_1(x_1,x_2,\ldots,x_k)$.

For example, function in Fig. 6b) represents function in Fig. 6a) without the loss of information, because for any assignment of {a,b}, the outputs of function in Fig. 6a) are compatible for all possible assignments of {a,b,c}. In particular, the output sets of Fig. 6a) for inputs (0,0,0) and (0,0,1) are compatible ({0} and {0,1}), and the output set of Fig. 6b) for inputs (0,0) is equal to 0, which is the result of merging of these two output sets. On the other hand, Fig. 6b) does not represent Fig. 7 without the loss of information, because there are two possible assignments of variables {a,b,c}, (1,1,0) and (1,1,1), such that in both of them {a,b} have values (1,1), while the corresponding output sets, {0} and {1}, are incompatible.

**Definition**. The problem of *input support minimization* for the MISF $\Re_1(x_1,x_2,\ldots,x_k)$ consists in finding the input variable subset $\{x_{i1},x_{i2},\ldots,x_{im}\}$ of minimum cardinality and MISF $\Re_2(x_{i1},x_{i2},\ldots,x_{im})$ that represents $\Re_1(x_1,x_2,\ldots,x_k)$ without the loss of information.

Observe that minimizing the input support leads to certain loss of *total information*, due to merging of compatible output sets and thereby restricting flexibility initially present in the function. In the above definitions, on the other hand, we speak about *essential information*, which should be preserved in decomposition unless we allow for some amount of “fuzziness”. The difference between these two types of information becomes clear after working with practical examples.

# 4 Approaches to Exact Minimization

## 4.1 General Strategy

In order to find the exact minimum of the number of input support variables, the following steps should be performed:

(1) Identify vacuous variables by exploring the BEMDD and forming the set of variables, on which the function depends; this set contains essential and inessential variables.

(2) Check each variable from the above set for being inessential. To achieve this, each variable is removed while letting other variables remain in the support, and then run procedure TestSupport() as described in Section 5. If the test succeeds (the reduced support does not involve losing information), this variable is inessential.

(3) Follow one of two complementary strategies that use branch-and-bound approach to find the exact minimum of the number of support variables:

**MinSupport1:** Initialize the support to all essential and inessential variables and try different sequences for *removing* inessential variables (the next variable to remove is selected heuristically). As long as removing a variable leads to a function that represents the given function without losing information, the set is a candidate for the minimum one.

**MinSupport2**: Initialize the support to all essential variables and try different sequences of adding inessential variables (the next variable to add is selected heuristically). As soon as *adding* a variable to the support leads to a function that represents the given function without losing information, this set is a candidate for the minimum one.

Both strategies use function TestSupport(), discussed in Section 5, to check whether the current support is enough to represent the function without the loss of information. Function TestSupport() does not explicitly derive cofactors with respect to the given variable. Instead, it uses a specialized BEMDD traversal to test the compatibility of cofactors implicitly. Notice that this algorithm is more efficient than the brute force approach based on first deriving all cofactors with respect to the given variable and then checking if the cofactors are indeed compatible.

The pseudo-code of procedures MinSupport1() and MinSupport2() is shown in Figs. 8a) and 8b), respectively.

Both procedures take three arguments: the MISF, the current variable support, and the set of inessential variables. First, large parts of the search space are pruned using the best solution found so far as an upper bound. Initially, the best solution is set to the number of variables in the given MISF before support minimization.

```
MinSupport1(bdd F, varset Support, varset Iness)
{
   // use the upper bound to prune the search
   if ( |Iness| = 0 or |Support|-|Iness| ≥ Best )
      return;

   // find the next variable to branch
   Var = GetVarWithMinimumValues( Iness );

   // test the support without this variable
   bdd ReducedF = TestSupport( F, Support-Var );

   if ( ReducedF != 0 ) // a new solution is found
   {
      // compare this solution with the best one
      Best = min( Best, |Support|-1 );
      // reduce the support and keep searching
      MinSupport1(ReducedF,Support–Var,Iness-Var);
   }
   else // a new solution is not found
   {
      // try removing other variables
      MinSupport1( F, Support, Iness-Var );
   }
}
```

Fig. 8a). Procedure implementing implicit support minimization based on *removing* inessential variables.

```
MinSupport2(bdd F, varset Support, varset Iness)
{
   // use the upper bound to prune the search
   if ( |Iness| = 0 or |Support| ≥ Best-1 )
      return;

   // find the next variable to branch
   Var = GetVarWithMaximumValues( Iness );

   // test the support with this variable
   bdd ReducedF = TestSupport( F, Support+Var );

   if ( ReducedF != 0 ) // a new solution is found
   {
      // compare this solution with the best one
      Best = min( Best, |Support|+1 );
   }
   else // a new solution is not found
   {
      // try other supports with this variable
      MinSupport2( F, Support+Var, Iness-Var );
      // try other supports without this variable
      MinSupport2( F, Support, Iness-Var );
   }
}
```

Fig. 8b). Procedure implementing implicit support minimization based on *adding* inessential variables.

In both procedures, a variable that is used to perform branching is determined. Selecting the branching variable is governed by attempts to achieve the minimum support size without taking into account the number of variables' values. Alternatively, it might be possible to formulate minimization procedure using different cost functions, for example, the sum total of values of the remaining support variables.

The function GetVarWithMinimumValues() selects the variable with the smallest number of values. The underlying motivation is to remove as many variables with a few values as possible, thereby creating a tight upper bound for later computation, when variables with more values are considered. Similarly, GetVarWithMaximumValues() finds the variable with the largest number of values.

The procedure MinSupport1() removes the selected variable from the support. MinSupport2(), on the other hand, adds the selected variable to the support. Both procedures test the resulting support for being large enough to represent the function without the loss of information. If the test succeeds, the support size is compared to the best solution found so far.

Finally, at the end of the pseudo-code, the functions are invoked recursively for sub-problems with variable sets, from which a branching variable is removed. In this way, both procedures keep track of the support variables that could not be removed and do not try removing them again in the same computational sub-tree.

## 5 Implicit Procedures

This section discusses implicit procedures, which constitute the main contribution of our paper. The first one is meant for testing whether the given support is enough to represent the given function without the loss of information. The second one is invoked from the first one to perform the recursive check whether two MISFs (in particular, cofactors) are compatible, and if yes, what is the result of their merging.

Because of the efficiency of the proposed representation of MISFs in solving various problems in decomposition, the usefulness of the procedures described in this section are not limited to input support minimization. The second procedure, for instance, can be used to efficiently check compatibility of cofactors, which constitutes an important step in iterative Curtis decomposition.

### 5.1 Traversal Routine for Support Testing

The pseudo-code for procedure TestSupport() is shown in Fig. 9. The procedure takes two arguments: the BEMDD and the input support to test. The return value is zero, if the support is not enough to represent the function without losing information. Otherwise, the return value is the BEMDD that represent the result of merging the cofactors with respect to variables not belonging to the support.

In the body of the function, the terminal cases are considered and the cache is checked for the already computed results. In this way, it is guaranteed that each node of the BEMDD is visited at most once.

```
bdd TestSupport( bdd F, varset Support )
{
   // consider the terminal cases
   if ( F = 0 or F = 1 )
      return F;
   // check the cache for the computed result
   if ( Result = CheckCacheForResult(F,Support) )
      return Result;

   // check whether this is an output value var
   if ( Var( F ) ∈ ValueVariables )
      return F;

   // solve the problem for cofactors and return
   // zero immediately if they are incompatible
   bdd F0 = TestSupport ( Low(F),  Support );
   if ( F0 = 0 )
      return 0;
   bdd F1 = TestSupport ( High(F), Support );
   if ( F1 = 0 )
      return 0;

   // consider two cases for the top-most variable
   if ( Var( F ) ∈ Support )
        Result = ITE( Var(F), F1, F0 );
   else /* if ( Var( F ) ∉ Support ) */
        Result = GetCompatible( F0, F1 );

   InsertIntoCache( F, Support, Result);
        return Result;
}
```

Fig. 9. Pseudo-code for function TestSupport().

The procedure calls function Var(), which returns the variable labeling the top-most node of the BEMDD representing F. If the variable returned belongs to the output-value-encoding variables, TestSupport() returns the MISF itself. As mentioned previously, the value encoding variables must be ordered at the bottom of the BEMDD.

The problem is solved for two sub-problems associated with cofactors of F with respect to the given binary variable. If an incompatibility is discovered during the solution of any of the sub-problems, the zero is returned immediately. In this way, it is guaranteed that the BEMDD is completely traversed only when the support is enough to represent the function.

Depending on the top-most variable, either a new node is created using the ITE-operator, or a call to the function checking compatibility of two sub-functions is performed. This behavior is explained by the following observation. If the given variable is one of the binary variables encoding the support, it remains in the BEMDD. On the other hand, if the binary variable does not encode the support, the sub-functions of this node should be checked for compatibility. This is done by the call to GetCompatible(), discussed in Section 5.2.

Notice that, because of this compatibility check, TestSupport() is an efficient implicit procedure. While performing the BEMDD traversal, TestSupport() checks the compatibility of the cofactors of F with respect to variables not included in the support. It does this without explicitly deriving the cofactors but rather by recursively checking compatibility of sub-functions rooted in the nodes labeled by the binary variables that encode multi-valued variables not included in the support.

### 5.2 Traversal Routines for Cofactor Compatibility

In this section, two procedures for checking cofactor compatibility are discussed. They are called from TestSupport() if the variable encountered in the BEMDD does not belong to those that encode the input support.

The pseudo-code of the first routine is given in Fig. 10.

```
bdd GetCompatible1( bdd F, bdd G )
{
   // treat terminal cases
   if ( F = 0 or G = 0 ) return 0;
   if ( F = 1 or G = 1 ) return F & G;

   if ( Result = CheckCacheForResult( F, G ) )
        return Result;

   // solve the problem depending on the position
   // of the top-most variables in F and G
   if ( Var(F)∈ValueVars and Var(G)∈ValueVars )
        Result = F & G;
   else if ( Var(F) = Var(G) )
   {
        C0 = GetCompatible1( Low(F),Low(G) );
         if ( C0 = 0 )
            return 0;
         C1 = GetCompatible1( High(F),High(G));
        if ( C1 = 0 )
            return 0;
        Result = ITE( Var(F), C1, C0 );
   }
   else if ( Var(F) < Var(G) )
   {
        C0 = GetCompatible1( Low(F), G );
        if ( C0 = 0 )
            return 0;
        C1 = GetCompatible1( High(F), G );
        if ( C1 = 0 )
            return 0;
        Result = ITE( Var(F), C1, C0 );
   }
   else /* if ( Var(F) > Var(G) ) */
   {
        C0 = GetCompatible1( F, Low(G) );
        if ( C0 = 0 )
            return 0;
        C1 = GetCompatible1( F, High(G) );
        if ( C1 = 0 )
            return 0;
        Result = ITE( Var(G), C1, C0 );
   }

   InsertIntoCache( F, G, Result);
   return Result;
}
```

Fig. 10. An efficient implementation of GetCompatible().

The structure of this routine mimics that of the APPLY operator available in the BDD package [10]. After considering the terminal cases and checking cache for results, two arguments of the function are expanded using the Shannon expansion with respect to the variable found top-most in the arguments' BEMDDs. The only difference compared to APPLY is that this function returns zero as soon as an incompatibility is discovered, without completing the graph traversal. In this way, computation time is saved when an incompatibility is discovered at the beginning of BEMDD traversal.

A simpler implementation of function GetCompatible(), using standard BDD operators, is possible (Fig. 11).

```
bdd GetCompatible2( bdd F, bdd G )
{
   // consider terminal cases
   if ( F = 0 or G = 0 )
        return 0;

   // compute the product of cofactors
   bdd Product = F & G;

   // if for some assignments of the variables
   // that are not in the support, the functions
   // do not have compatible output assignments,
   // return 0; otherwise, return the product
   if ( ∃ ValueVars (Product) != 1 )
        return 0;
   else
        return Product;
}
```

Fig. 11. A simple implementation of GetCompatible().

This implementation relies on existential quantification to check whether the product of F and G depends on the value variables for all assignments of variables not included in the support. If it does depend on the value variables, then F and G are compatible and their product is the result of merging them. If F and G are incompatible, then zero is returned, even though their product is not zero.

The following example illustrates this computation. Consider the function with the truth table is given in Fig. 7. Using the binary variables as the input support encoding variables, the following is found:

$$F = \bar{a}\bar{b}\bar{c}\bar{v} + \bar{a}\bar{b}c + \bar{a}b\bar{c} + \bar{a}bc\bar{v} + ab\bar{c}v + abc\bar{v} + a\bar{b}\bar{c} + a\bar{b}c\bar{v}\ .$$

The columns are represented by the following functions:

$$F_0 = \bar{a}\bar{b}\bar{v} + \bar{a}b + abv + a\bar{b}$$

$$F_1 = \bar{a}\bar{b} + \bar{a}b\bar{v} + ab\bar{v} + a\bar{b}\bar{v}\ .$$

Their product is

$$P = F_0 \,\&\, F_1 = \bar{a}\bar{b}\bar{v} + \bar{a}b\bar{v} + a\bar{b}\bar{v}\ .$$

As a result of quantification, the following is found:

$$\exists v\,( \bar{a}\bar{b}\bar{v} + \bar{a}b\bar{v} + a\bar{b}\bar{v} ) = \bar{a}\bar{b} + \bar{a}b + a\bar{b} = \bar{a} + \bar{b} \neq 1,$$

which means that the cofactors of F with respect to c are not compatible. It is not possible to remove c from the support of F.

It is reasonable to expect that the second implementation is less efficient compared to the first one because the quantification operator traverses the BEMDD completely even if the incompatibility of F and G is discovered right away. In practice, however, input support minimization based on the second implementation turned out to be 20% faster.

We assume that this speed-up is because existential quantification, being one of the standard operators, is efficiently implemented in the BDD package, by directly manipulating nodes in the node table. Our implementation of GetCompatible1(), on the other hand, loses some of its efficiency due to multiple calls to functions Var(), Low(), High(), and GetCompatible().

## 6 Deriving an Approximate Measure

In this section, the following issue is addressed. Given an MISF, determine which strategy works better: removing inessential variables from the support, or adding essential variables to the support. Assuming that removing one variable and adding one variable are computationally equivalent, the first strategy is more efficient when less than one half of all variables are inessential and can be removed. Vice versa, the second strategy is more efficient when it is possible to remove more than one half of inessential variables.

Below, an approximate measure is derived for the case set size (the number of minterms, in which MISF is specified), for the following statement to be true: "A half of the input variables are inessential and can be removed from the support".

Suppose F is a random k-valued-output n-variable function, all input of which are k-valued, and C is its care set size.

Initially, the domain of F consists of $k^n$ minterms (vertices) and the density of the care set minterms is $C/k^n$. When one half of the variables are removed, the domain shrinks to $k^{n/2}$ minterms.

Suppose one half of the input variables have already been removed from the support, and now the next inessential variable is being removed. The variable can be removed if all k cofactors of F with respect to this variable are compatible.

Two cofactors are compatible if care minterms of the first cofactor do not overlap with care minterms of the second cofactor. Suppose there are X care minterms in the first cofactor and Y care minterms in the second cofactor. Then, assuming that $\mu$ is the density of care minterms, the cofactors are compatible with the probability

$$P_1(X,Y) = (1-\mu*X)^Y \approx 1 - \mu*X*Y.$$

Approximation in this formula is based on the assumption that the density of care minterms ($\mu$) is very small, which is true for weakly-specified functions and relations.

Because the compatibility of each cofactor with the product of other cofactors is an independent event, the resulting probability is the product of partial probabilities. Assuming that a cofactor's domain, on average, contains $N = C/k$ care minterms and the density of care minterms after removing a half of the variables is $\mu = C/k^{n/2}$, the following is found:

$$P = P_1( N,N ) * P_1( 2*N,N ) *\ldots* P_1( (k-1)*N,N ) \approx$$
$$\approx [P_1( k*N/2, N )]^{k-1} = [1 - \mu*k*N^2/2]^{k-1} \approx$$
$$\approx 1 - \mu*k^2N^2/2 = 1 - C^3/k^{n/2}/2.$$

The condition when the variable can be removed with the probability 1/2 is

$$1 - C^3/k^{n/2}/2 = 1/2.$$

For this to be true, the following should hold

$$C = k^{n/6}.$$

To derive this formula, an assumption was made that F is a function. Let us note that, when dealing with random relations (which can have more than one output value for each input value assignment), it is more probable that the randomly generated care minterms are compatible than it was assumed in the above approximation. So, for a random relation, formula $C = k^{n/6}$ gives an upper bound on the care set size, for which the second strategy (adding variables) is more efficient.

Consider an example. For a weakly-specified relation depending on twenty five-valued variables ($n = 20$, $k=5$), the size of the care set should be less than $5^{20/6} \approx 212$ minterms for the second strategy to be more efficient.

## 7 Experimental Results

Algorithms described above have been implemented in a C++ program. The results of running the program using the set of POLO benchmarks [9] on a Windows 98 computer, with Pentium-II CPU 266MHz and 64M RAM, are given in Table 1.

The first column lists names of the benchmark functions. Column **In#** gives the number of multi-valued input variables. Column **Val#** gives the total sum of values of input variables for each function.

Columns **Node#** and T1 give the number of nodes in the BEMDD for the given function and the time (in seconds) needed to build the BEMDD.

Columns **Sift#** and T2 give the number of nodes in the BEMDD after variable reordering using the sifting algorithm and the time needed to perform the reordering.

Columns **IE#** and T3 give the number of inessential variables and the time needed to determine them.

Column **MS#** gives the number of variables in the minimum support for the given benchmark.

Column **Iter1#** gives the number of calls to function TestSupport() when the first strategy is used (removing inessential variables from the support). Column T4 gives the running time of procedure MinSupport1().

Column **Iter2#** gives the number of calls to function TestSupport() when the second strategy was used (adding inessential variables to the support). Column T5 gives the running time of procedure MinSupport2().

Dashes in some of the columns of Table 1 mean that all the variables of these functions are essential and, therefore, the procedures for input support minimization have not been applied to them.

In the last row of Table1, the average of the parameter in each column is given. For the last five columns of the table, the average does not include measurements for the benchmarks “Flag” and “Mushroom”. The running time for these two benchmarks shows the NP-completeness of the problem of exact input support minimization.

| **Benchmark** | **In#** | **Val#** | **Node#** | **T1, c** | **Sift#** | **T2, c** | **IE#** | **T3, c** | **MS#** | **Iter1#** | **T4, c** | **Iter2#** | **T5, c** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Balance | **4** | 20 | 91 | 0.05 | 85 | 0.93 | **0** | 0.01 | **-** | - | - | - | - |
| Breastc | **9** | 90 | 3903 | 0.33 | 3490 | 3.08 | **8** | 0.83 | **5** | **64** | 2.47 | 94 | 3.51 |
| Bridges1 | **9** | 29 | 370 | 0.05 | 304 | 0.99 | **0** | 0.05 | **-** | - | - | - | - |
| Bridges2 | **10** | 32 | 463 | 0.06 | 354 | 1.21 | **1** | 0.05 | **9** | 1 | 0.01 | 1 | 0.01 |
| Chess1 | **6** | 40 | 7866 | 6.10 | 6367 | 2.42 | **0** | 0.17 | **-** | - | - | - | - |
| Cloud | **6** | 48 | 557 | 0.01 | 451 | 1.16 | **1** | 0.06 | **5** | 1 | 0.01 | 1 | 0.01 |
| Employ2 | **7** | 29 | 66 | 3.41 | 51 | 0.99 | **0** | 0.05 | **-** | - | - | - | - |
| Flag | **28** | 133 | 5937 | 0.44 | 4887 | 6.2 | **28** | 4.67 | **7?** | 2890 | >300 | 1951 | >300 |
| Flare1 | **10** | 33 | 349 | 0.28 | 298 | 1.1 | **2** | 0.06 | **8** | 2 | 0.01 | 1 | 0.01 |
| Monks1tr | **6** | 17 | 124 | 0.05 | 118 | 0.87 | **3** | 0.01 | **3** | 3 | 0.01 | 1 | 0.01 |
| Monks2tr | **6** | 17 | 143 | 0.06 | 138 | 0.88 | **0** | 0.01 | **-** | - | - | - | - |
| Monks3tr | **6** | 17 | 122 | 0.06 | 119 | 0.88 | **2** | 0.01 | **4** | 2 | 0.01 | 1 | 0.01 |
| Mushroom | **22** | 117 | 1253 | 6.32 | 894 | 3.41 | **22** | 0.71 | **4** | 2879 | 58.44 | 5765 | 28.34 |
| Post-oper | **8** | 23 | 187 | 0.06 | 161 | 0.94 | **0** | 0.01 | **-** | - | - | - | - |
| Programm | **12** | 42 | 17495 | 8.56 | 16330 | 6.75 | **0** | 1.65 | **-** | - | - | - | - |
| Sensory | **11** | 36 | 1780 | 0.22 | 1111 | 1.32 | **10** | 0.49 | **5** | **192** | 3.41 | 194 | 2.47 |
| Sleep | **9** | 83 | 802 | 0.06 | 697 | 1.71 | **7** | 0.16 | **5** | **30** | 0.38 | 32 | 0.22 |
| Tic-tac-toe | **9** | 27 | 576 | 0.27 | 536 | 1.15 | **9** | 0.16 | **8** | **44** | 0.17 | 503 | 1.21 |
| Trains | **32** | 105 | 318 | 0.06 | 235 | 2.75 | **32** | 0.27 | **1** | 32 | 0.17 | **4** | 0.01 |
| Zoo | **16** | 39 | 285 | 0.06 | 216 | 1.10 | **14** | 0.11 | **5** | **118** | 0.38 | 328 | 0.72 |
| alet | **18** | 180 | 19363 | 2.20 | 18968 | 21.15 | **10** | 7.03 | **9** | **10** | 3.51 | 46 | 6.98 |
| c3a | **14** | 46 | 203 | 0.01 | 160 | 1.31 | **14** | 0.05 | **2** | 80 | 0.22 | **52** | 0.17 |
| c3b | **14** | 48 | 229 | 0.01 | 197 | 1.37 | **14** | 0.11 | **3** | 397 | 0.87 | **109** | 0.39 |
| c6a | **13** | 61 | 282 | 0.01 | 247 | 1.59 | **13** | 0.11 | **2** | 78 | 0.22 | **17** | 0.06 |
| c6b | **13** | 66 | 271 | 0.01 | 239 | 1.54 | **13** | 0.06 | **2** | 87 | 0.27 | **15** | 0.11 |
| d4 | **14** | 55 | 461 | 0.01 | 402 | 1.49 | **14** | 0.17 | **3** | 201 | 0.98 | **120** | 0.33 |
| d6 | **13** | 87 | 615 | 0.06 | 552 | 1.93 | **13** | 0.16 | **2** | 80 | 0.61 | **15** | 0.05 |
| d8 | **32** | 166 | 1369 | 0.05 | 1271 | 6.81 | **32** | 1.04 | **2** | 500 | 6.98 | **34** | 0.38 |
| **Average** | **12.6** | **60.2** | **2197** | **0.98** | **1973** | **2.63** | **8.7** | **0.50** | **4.3** | **101** | **1.09** | **83** | **0.87** |

Table 1. Experimental results using POLO benchmarks.

## 8 Conclusion

Described in this paper is an improved representation of incompletely specified functions and relations based on BEMDDs and input support minimization algorithms, with potential application in machine learning and data mining.

The main contributions are:

We introduced a new representation of MISFs using BEMDDs that are different from other decision diagram based representations used in the literature [2,4]. This representation allows for an efficient solution of a wide range of computationally intensive tasks in functional decomposition.

In particular, the problem of input support minimization has been addressed. We propose two algorithms that are neither generalization to MV logic of the algorithm from [7] nor BDD based implementations of the methods presented in [5,6].

In forthcoming publications, we will show how BEMDDs can be used to solve the column compatibility problem with large bound sets and implement multi-valued bi-decomposition.

## Acknowledgements

The work on multi-valued decomposition for robotics applications is partially supported by an Intel grant for development of undergraduate robotics laboratories.

Our implementation of support minimization is based on state-of-the-art Binary Decision Diagram package BuDDy, Release 1.7 [10], by Jorn Lind-Nielsen, Department of Information Technology, Technical University of Denmark. Finally, we are grateful to Forrest Brewer from University of California in Santa Barbara, who pointed out a connenction of our research to that of Bill Lin [7].

## References

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

2 Y.T.Lai, M. Pedram, S.B.K.Vrudhula. BDD-based Decomposition of logic functions with applications to FPGA synthesis. *Proc. of DAC '93*, pp. 642-647.

3 M.Perkowski, M.Marek-Sadowska, L.Jozwiak, T.Luba, S.Grygiel, M.Nowicka, R.Malvi, Z.Wang, J.S.Zhang. Decomposition of multiple-valued relations. *Proc. of ISMVL'97,* pp. 13-18.

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

5 L. Jozwiak, P. A. Konieczny. Heuristic Algorithms for Minimal Input Support Problem. *Workshop on Design Methodologies for Microelectronics*, 1995, pp. 196-205.

6 L. Jozwiak, N.Ederveen. Genetic Algorithm for Input Support Selection. *Proc. of ICCIMA '98 (Intl. Conf. on Comp. Intelligence and Multim. Applications),* pp. 765-772.

7 B.Lin. Efficient Symbolic Support Manipulation. *Proc. of ICCD '93*, pp. 513-516.

8 C. Files. *A New Functional Decomposition Method as Applied to Machine Learning and VLSI Design.* Ph.D. thesis, Portland State University, June 2000.

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

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