

Generalized Symmetries in Boolean Functions: Fast Computation and Application to Boolean Matching¹

Jin S. Zhang¹ Malgorzata Chrzanowska-Jeske¹ Alan Mishchenko² Jerry R. Burch³

¹ Department of ECE, Portland State University, Portland, OR

² Department of EECS, UC Berkeley, Berkeley, CA

³ Synopsys Inc. Hillsboro, OR

Abstract

In recent years, the notion of symmetry has been extended from classical symmetries to also include constant cofactor symmetries, single variable symmetries and Kronecker symmetries. All these symmetries form a generalized symmetry scheme. Existing methods to detect generalized symmetries require computing the cofactors for each pair of variables to check certain relationships between the cofactors. In this paper, we present a new algorithm that detects all pairs of symmetric variables in one pass over a multi-output BDD. Experiments on the MCNC benchmarks are encouraging. We also propose a potential application of generalized symmetries in Boolean matching. *Keywords:* Kronecker symmetries, generalized symmetries, Boolean matching.

1. Introduction

Classical symmetries have been well studied and used in many applications, such as functional decomposition in technology independent logic synthesis [1-4], technology mapping [5][6][23], BDD minimization [7], testing [8][9] and verification [10]. Several algorithms [11-14] have been proposed to detect classical symmetries more efficiently.

Classical symmetries and single variable symmetries are based on the relationship of two cofactors. Constant cofactor symmetries indicate that one of the cofactors is a constant function. Kronecker symmetries [15][16] are based on the relationships among three or four cofactors. The generalized symmetry scheme contains 30 types of *skew* and *non-skew* symmetries. Table 1 lists all the 30 types of symmetries, the relationships between the cofactors and its symbol representation.

Table 2 gives the number of symmetries in a selection of MCNC benchmark functions. Each cell contains the number of *non-skew* and *skew* symmetries. The last two rows report the total number and the percentage of symmetries in the functions.

Table 2 shows that:

- Generalized symmetries are very common.
- There are many more *non-skew* symmetries than *skew* symmetries.
- The often used classical symmetry is the least common form of symmetries in these circuits. More improvement could be made if other forms of symmetry are also taken into consideration.

Table 1. Generalized Symmetries

#	Property (non-skew/skew)	Name	Symbol
1	$F_{00} = 0/1$	Constant Cofactor Symmetries	C_0
2	$F_{01} = 0/1$		C_1
3	$F_{10} = 0/1$		C_2
4	$F_{11} = 0/1$		C_3
5	$F_{10} \oplus F_{01} = 0/1$	Classical Symmetries	T_1 (NE)
6	$F_{00} \oplus F_{11} = 0/1$		T_2 (E)
7	$F_{00} \oplus F_{01} = 0/1$	Single Variable Symmetries	T_3
8	$F_{10} \oplus F_{11} = 0/1$		T_4
9	$F_{00} \oplus F_{10} = 0/1$		T_5
10	$F_{01} \oplus F_{11} = 0/1$		T_6
11	$F_{01} \oplus F_{10} \oplus F_{11} = 0/1$	Kronecker Symmetries	K_0
12	$F_{00} \oplus F_{10} \oplus F_{11} = 0/1$		K_1
13	$F_{00} \oplus F_{01} \oplus F_{11} = 0/1$		K_2
14	$F_{00} \oplus F_{01} \oplus F_{10} = 0/1$		K_3
15	$F_{00} \oplus F_{01} \oplus F_{10} \oplus F_{11} = 0/1$		K_4

Table 2. Number of Symmetries in MCNC Benchmark

Name	Constant Cofactor	Classical	Single Variable	Kronecker
9symml	0 / 0	36 / 0	0 / 0	0 / 0
alu2	27 / 3	7 / 1	15 / 6	18 / 10
alu4	39 / 3	11 / 1	31 / 6	48 / 6
apex6	1725 / 39	408 / 16	2540 / 29	906 / 42
term1	589 / 117	70 / 2	510 / 0	119 / 15
b1	8 / 2	5 / 3	0 / 4	3 / 11
b9	219 / 156	81 / 8	464 / 9	51 / 24
f51m	4 / 4	5 / 8	3 / 67	32 / 5
x1	872 / 497	300 / 6	1696 / 7	188 / 66
c8	61 / 24	114 / 16	343 / 25	233 / 49
i1	111 / 43	103 / 14	268 / 18	39 / 21
z4ml	1 / 1	22 / 5	0 / 52	22 / 6
too_large	234 / 0	20 / 0	482 / 0	2 / 0
t481	0 / 0	8 / 0	16 / 0	80 / 0
cc	191 / 5	44 / 3	166 / 6	40 / 10
k2	1681 / 0	559 / 0	4191 / 0	3673 / 0
frg1	7 / 21	7 / 3	101 / 4	44 / 8
rot	1658 / 365	540 / 93	5216 / 99	574 / 142
pml	176 / 147	85 / 1	255 / 2	33 / 32
Total	7603 / 1427	2425 / 180	16297 / 334	6105 / 447
Ratio	22% / 4%	7% / 0.5%	47% / 1%	18% / 1%

¹ This work was supported in part by the NSF grant CCR-9988402

The naïve way of detecting generalized symmetries involves computing the cofactors for each pair of variables, then checking to see if the cofactors satisfy the relationships specified in Table 1. This approach is straightforward, yet quite inefficient for large circuits. In this paper, we propose a more efficient algorithm that detects the symmetric variables recursively. The algorithm, presented in the following sections, is an extension of [14], which only detects classical symmetries. Constant cofactor symmetries, single variables symmetries and Kronecker symmetries requires different consideration in the recursive steps from that of the classical symmetries.

Classical symmetries have been used in many applications in electronic design automation. It is natural to see how generalized symmetries can be applied to those applications to better the end results. In the paper, we prove the applicability of generalized symmetries as signatures in Boolean matching.

The remainder of this paper is organized as follows. We introduce the background information and the theorem on which the algorithm is based in section 2. Section 3 discusses the core algorithm. In section 4, we touch upon a few implementation issues. Experimental results are given in section 5. We present one potential application of generalized symmetries in section 6. Section 7 concludes the paper.

2. Background

The discussions in this paper are based upon completely specified Boolean functions and Boolean variables.

The variables that a function F depends on are called the *support* of F , denoted by $supp(F)$.

A *cofactor* of a function $F(\dots, x_i, \dots, x_j, \dots)$ with respect to (w.r.t.) variables x_i and x_j , is the function resulting from F when x_i and x_j are substituted with specific values. For example, the cofactor of F w.r.t. $x_i = 0$ and $x_j = 0$, is the function $F[x_i \leftarrow 0, x_j \leftarrow 0]$, which is denoted by F_{00} .

The Shannon expansion of a Boolean function F is: $F = x \cdot F_x + x' \cdot F_{x'}$, where F_x and $F_{x'}$ are the positive and negative cofactors of function F , w.r.t. variable x .

Two variables, x_i and x_j , of function $F(\dots, x_i, \dots, x_j, \dots)$ are symmetric if the function remains invariant when the variables are swapped, i.e. $F(\dots, x_i, \dots, x_j, \dots) = F(\dots, x_j, \dots, x_i, \dots)$. This is the earliest notion of *symmetry*. For any pair of variables x_i and x_j , there are four cofactors, $F_{00}, F_{01}, F_{10}, F_{11}$. This symmetry translates to the relationship of the cofactors that $F_{01} = F_{10}$ or equivalently $F_{01} \oplus F_{10} = 0$, as shown in Table 1. This symmetry is also called the *non-skew non-equivalent symmetry*[20]. *Non-skew equivalent symmetry* exists when $F_{00} = F_{11}$. These are called classical symmetries. Skew classical symmetries exist when the two cofactors are complement of each other. Constant cofactor symmetries exist when one of the cofactors is a constant function. Single variable symmetries imply that one of the cofactors of the function w.r.t. a variable does not depend on the other variable. For example, $F_{00} = F_{01}$ means that the negative cofactor of F w.r.t variable x_i does not depend on x_j .

Kronecker symmetries are derived from Davio expansions [24] and Kronecker Functional Decision Diagrams [21][22]. The positive Davio expansion for a Boolean function F is: $F = F_x \oplus x \cdot (F_x \oplus F_{x'})$. The negative Davio expansion is: $F = F_x \oplus x' \cdot$

$(F_x \oplus F_{x'})$. A Kronecker Function Decision Diagram (KFDD) allows Shannon and both Davio expansions in generating the decision diagram, although only one type of expansion is allowed per variable.

Definition 1. Let $F(x_1 \dots x_n)$ be a Boolean function. Let i and j be integers between 1 and n , inclusive. Let $g = \langle g_0, g_1, g_2, g_3, g_4 \rangle$ be a Boolean vector of length 5. At least one of g_0, g_1, g_2, g_3 is 1. We say that variables x_i and x_j have generalized symmetry g in function F (in symbols, $G(F, (i, j), g)$) iff:

$$\forall x_1 \dots x_n [g_0 F_{00} \oplus g_1 F_{01} \oplus g_2 F_{10} \oplus g_3 F_{11} = g_4].$$

For example, variables x_i and x_j having *non-skew T1* classical symmetry is equivalent to x_i and x_j having generalized symmetry $g = \langle 0, 1, 1, 0, 0 \rangle$.

Definition 2. *Symmetry type* is defined to be each unique assignment of g .

For example, $g = \langle 1, 1, 0, 1, 0 \rangle$ is a unique *symmetry type*, corresponding to *non-skew Kronecker K2* symmetry. $g = \langle 1, 1, 0, 1, 1 \rangle$ is another unique *symmetry type*, corresponding to *skew Kronecker K2* symmetry. There are 30 symmetry types in generalized symmetries.

Definition 3. *Symmetry category* is defined to include all the symmetry types such that the sum of g_0, g_1, g_2, g_3 are the same.

For example, symmetry $\langle 0, 1, 1, 0, 0 \rangle$ and $\langle 1, 0, 0, 1, 0 \rangle$ belong to the same *symmetry category*, $\langle 1, 0, 0, 0, 0 \rangle$ and $\langle 1, 1, 1, 0, 0 \rangle$ belong to different *symmetry category*. There are 4 symmetry categories in generalized symmetries.

The fast computation algorithm proposed in this paper is based on the following theorem.

Theorem 1. Let F be a function and x_i, x_j, x_k be three variables in the *support* of F . Function F is symmetric in x_i, x_j with any type of generalized symmetries if and only if both cofactors of F w.r.t. x_k are symmetric in x_i and x_j with the same symmetry.

Proof: x_i and x_j have symmetry g in function F iff:

$$\forall x_1 \dots x_n [g_0 F_{00} \oplus g_1 F_{01} \oplus g_2 F_{10} \oplus g_3 F_{11} = g_4].$$

This equation holds true for all values of x_k , where $1 \leq k \leq n, k \neq i$ and $k \neq j$. That is:

$$\forall x_1 \dots x_n [g_0 F_{00}[x_k \leftarrow 0] \oplus g_1 F_{01}[x_k \leftarrow 0] \oplus g_2 F_{10}[x_k \leftarrow 0]$$

$$\oplus g_3 F_{11}[x_k \leftarrow 0] = g_4], \text{ and}$$

$$\forall x_1 \dots x_n [g_0 F_{00}[x_k \leftarrow 1] \oplus g_1 F_{01}[x_k \leftarrow 1] \oplus g_2 F_{10}[x_k \leftarrow 1]$$

$$\oplus g_3 F_{11}[x_k \leftarrow 1] = g_4].$$

Therefore x_i and x_j have the same generalized symmetry g in both cofactors of F w.r.t. x_k . The reverse implication holds because x_k must be 0 or 1. \square

We use the symmetry graph to represent the computed symmetries as a set of symmetric variable pairs. The vertices of the symmetry graph correspond to variables in the *support* of the function. The edges connect symmetric variable pairs. The graph operations *union* (\cup) and *intersection* (\cap) are similarly defined as the set *union* and *intersection* on the sets of edges. The *Cartesian product* (\times) of variable x by a set of variables Y results in a graph composed of edges connecting the vertex of variable x with vertices of variables in Y . The symmetry graph is represented by Zero-Suppressed Decision Diagrams (ZDD), a canonical representation of the symmetry information [14].

3. Symmetry computation algorithm

3.1 Computational core

Theorem 1 states that we can compute the symmetries of a function if we know the symmetries of the function's cofactors w.r.t. a variable. Therefore we can recursively solve the sub-problems and derive the final solution from the partial solutions. Figure 1 is the pseudo code of the recursive core of the algorithm. It takes a function F , a set of variables V , $skewT$ and $symmT$ as inputs. F is the function whose symmetries are being computed. V is initialized to be the *support* of F at the top level, but could become a super set of F 's *support* in the subsequent recursive calls. $skewT$ and $symmT$ are integers indicating the type of symmetry to be computed. The program returns the symmetry graph representing pairs of symmetric variables.

```

symgraph ComputeSymmetries_rec(func  $F$ , varset  $V$ , int  $skewT$ , int  $symmT$ )
{
  Step 1: if ( $F$  is a constant function)
    return ConstantHandling ( $F$ ,  $V$ ,  $skewT$ ,  $symmT$ );
  Step 2:  $x = \text{supp}(F)$ ;
    ( $F_0, F_1$ ) = Cofactors ( $F, x$ );
  Step 3:  $RemVars = \text{supp}(F) - x$ ;
     $S_0 = \text{ComputeSymmetries\_rec}$  ( $F_0, RemVars, skewT, symmT$ );
    if ( $S_0 = \emptyset$ )
       $S_1 = \emptyset$ ;
    else
       $S_1 = \text{ComputeSymmetries\_rec}$  ( $F_1, RemVars, skewT, symmT$ );
  Step 4:  $Y = \text{SymmetricVars\_rec}$  ( $F_0, F_1, RemVars, skewT, symmT$ );
     $S_2 = x \times Y$ ;
     $S_3 = \text{SymmForSkippedVars}$  ( $F, V, skewT, symmT$ );
     $S = (S_0 \cap S_1) \cup S_2 \cup S_3$ ;
  Step 5: return  $S$ .
}

```

Figure 1. Pseudo-code of symmetry computation core

The recursive steps are the same for different symmetry categories. However the handling of Step 1 and S_3 computation in step 4 are different due to the difference in cofactor relationships of each symmetry. The following paragraphs explain each step in the pseudo code. Discussions on classical symmetries are included here for completeness, though details can be found in [14].

Step 1. Procedure *ComputeSymmetries_rec* checks to see if the function F is a constant. The constant function is handled differently depending on the symmetry types.

Constant cofactor symmetries: If F is constant 0, then all the variables in V are symmetric with each other with *non-skew* constant cofactor symmetries C0 to C3. Similarly, if F is constant 1, then all the variables in V are symmetric with each other with *skew* C0 and C3 symmetries.

Classical symmetries: If F is a constant, then all the variables in V are symmetric with each other with *non-skew* classical symmetries T1 and T2.

Single variable symmetries: If F is a constant, then all the variables in V are symmetric with each other with *non-skew* T3 to T6 single variable symmetries.

Kronecker symmetries: If F is constant 0, then all the variables in V are symmetric with each other with *non-skew*

Kronecker K0 to K4 symmetries. If F is a constant 1, the variables in V are symmetric with each other with *non-skew* K4 symmetry and *skew* K0 to K3 symmetries. The reason is: a constant 1 function has all its four cofactors as constant 1 as well. Any 3 cofactors being EXOR-ed together will result in constant 1, which satisfies the requirements for *skew* Kronecker symmetries K0 to K3. All four cofactors being EXOR-ed together will result in constant 0, which satisfies the requirement for *non-skew* K4 symmetry.

In short, when F is a constant, the program returns a complete symmetry graph or an empty set depending on the value of the function, the symmetry, and skew type.

Step 2. Compute the *support* of F and cofactor the function w.r.t. one variable in its *support*. This results in two different Boolean functions F_0, F_1 . These are constant time operations when BDDs are used to represent the function.

Step 3. This is where the recursive calls are happening. First *ComputeSymmetries_rec* is called with the negative cofactor F_0 and the variables set which equals to the *support* of F without the cofactoring variable x . If there is no symmetry in the negative cofactor F_0 , then we don't need to check for its positive cofactor, according to Theorem 1. As a result, the computation is very efficient when there are no symmetries in the design. Next, the same procedure is called to compute the symmetries for the positive cofactor.

Step 4. The first part of the solution comes from the *intersection* of partial solution S_0 and S_1 , because, according to Theorem 1, only the symmetries of both cofactors are symmetries of the function.

Partial solution S_2 contains the symmetries that involve the cofactoring variable x . This is computed through another recursive procedure *SymmetricVars_rec*, which will be discussed later.

The set S_3 contains symmetries that involve variables that are in V but not in the *support* of F . This happens when the cofactors of the function do not depend on all the variables in the initial *support* of F minus the cofactoring variables. Figure 2 is a fragment of the BDD, which illustrates the scenario. The negative cofactor of F w.r.t. variable a only depends on variables c, d and e . Therefore in the recursive procedure, variables b is skipped over. Hence we need to detect symmetries involving these "skipped" variables. This is where the differences come in for different symmetry categories.

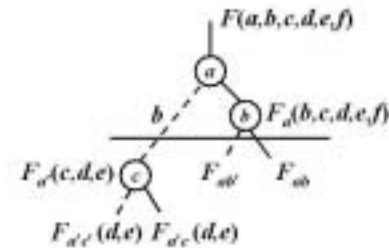


Figure 2. Illustrations of skipped variables

Constant cofactor symmetries: All the skipped variables have *non-skew* C0 and C2 symmetry with the variables in the *support* of F if the negative cofactors of F w.r.t. these variables are 0. Similarly they have *non-skew* C1 and C3 symmetries if the positive cofactors of F are 0. Skew symmetries require the value of the cofactors be 1. For example,

in Figure 2, if $F_{a'c} = 0$, then $F_{a'b'c} = 0$. Therefore variables b and c have *non-skew* C0 symmetry in the negative cofactor of F w.r.t. variable a .

Classical symmetries: All the skipped variables (including those variables not in the *support* of F) are symmetric with each other with *non-skew* classical T1 and T2 symmetries. For skew symmetries, the skipped variables are symmetric with all the variables in the *support* of F whose negative and positive cofactors are complements of each other.

Single Variable symmetries: All the skipped variables are symmetric with each other with *non-skew* T3 to T6 symmetries. The skipped variables also have *non-skew* T5 and T6 symmetries with the variables in the *support* of F . The skipped variables have *skew* T5 and T6 single variable symmetries with all the variables in the *support* of F whose negative and positive cofactors are complements of each other.

Kronecker symmetries: All the skipped variables are symmetric with each other and with the variables in the *support* of F with *non-skew* K4 symmetry. For example, in Figure 2, variables b and f are not in the support of F_a . This implies that $F_{a'b'f} = F_{a'bf} = F_{a'bf'} = F_{a'bf}$. The EXOR of these 4 equivalent cofactors is always constant 0. Therefore b and f have *non-skew* K4 symmetry in the negative cofactor of F w.r.t. variable a . However they do not have *non-skew* K0 to K3 symmetries because these symmetries require the EXOR of only 3 equivalent cofactors. For *non-skew* K0 to K3 symmetries, the checking is a little bit more complicated. We will take K0 symmetry as an example. In Figure 2, to check whether variables b and d are symmetric, we need to compute whether $F_{a'b'd} \oplus F_{a'bd} \oplus F_{a'bd}$ is constant 0. This translates to checking whether $F_{a'd} \oplus F_{a'd} \oplus F_{a'd}$, or equivalently, $F_{a'd}$ is 0. In short, checking whether the skipped variables have K0 to K3 symmetry with the variables in the *support* of F requires checking whether the appropriate cofactor of F w.r.t. these variables are constant 0 for *non-skew* symmetries or 1 for *skew* symmetries.

The complete solution of S is equal to the *union* of S_2, S_3 , and with the *intersection* of S_0 and S_1 ($S = (S_0 \cap S_1) \cup S_2 \cup S_3$).

3.2 Detecting variables symmetric with a variable

Procedure *SymmetricVars_rec* takes F_0 and F_1 , the two cofactors of F w.r.t. variable x , the set of candidate variables Y and *skewT*, *symmT* as inputs. It returns the subset of Y such that, for each variable in this subset, it is symmetric with x in the original function F . Figure 3 is the pseudo-code for this procedure. (G, H are used to represent the cofactors F_0 and F_1 hereafter to avoid multiple indexing)

Once again, the program differs in step 1, R_2 computation in step 4 and step 5 for different symmetries.

Step1. Procedure *SymmetricVars_rec* checks to see if either G or H is a constant function. The constant function is handled differently depending on the symmetry categories.

Constant cofactor symmetries: If G is constant 0, then variable x has *non-skew* C0 and C1 constant cofactor symmetries with the variables in Y . If H is constant 0, then variable x has *non-skew* C2 and C3 constant cofactor symmetries with the variables in Y . Skew symmetries require the corresponding value of G or H to be constant 1.

Classical symmetries: If G and H have the same constant value, then variable x has *non-skew* T1 and T2 classical symmetries with the variables in Y . If G is complement of H , then x has *skew* classical symmetries with the variables in Y .

Single variable symmetries: If G is a constant, then variable x has *non-skew* T3 symmetries with the variables in Y . Similarly, if F is a constant, then it has *non-skew* T4 symmetries with the variables in Y . If G and H have the same constant value, then variable x has *non-skew* T5 and T6 single variable symmetries with the variables in Y . If G and H are complements of each other, then it has *skew* T5 and T6 single variable symmetries with the variables in Y .

```

varset SymmetricVars_rec(func  $G$ , func  $H$ , varset  $Y$ , int  $skewT$ , int  $symmT$ )
{
  Step 1: if ( $G$  or  $H$  is a constant function )
    ConstantHandling ( $G, H, Y, skewT, symmT$ );
  Step 2:  $z = \text{Var}(Y)$ 
    ( $G_0, G_1$ ) = Cofactors ( $G, z$ );
    ( $H_0, H_1$ ) = Cofactors ( $H, z$ );
  Step 3:  $R_0 = \text{SymmetricVars\_rec}(G_0, H_0, Y-z, skewT, symmT)$ ;
    if ( $R_0 = \emptyset$ )
       $R_1 = \emptyset$ ;
    else
       $R_1 = \text{SymmetricVars\_rec}(G_1, H_1, Y-z, skewT, symmT)$ ;
  Step 4:  $R_2 = \text{SkippedSymmetricVars}(G, H, Y, skewT, symmT)$ ;
       $R = (R_0 \cap R_1) \cup R_2$ ;
  Step 5: if (SatisfyCofRelations ( $G_0, G_1, H_0, H_1, skewT, symmT$ ))
       $R = R \cup z$ ;
  Step 6: return  $R$ .
}

```

Figure 3. Computing the set of variables that are symmetric with a variable

Kronecker symmetries: What type of symmetry exists depends on the combinations of the constant values that G and H take. The columns of Table 3 list the four constant combinations that G and H can assume. The rows correspond to the possible symmetry types. The symbol “S” in the cell indicates that the type of symmetry on the row exists under the particular combination of G and H on the column. For example, when G and H have constant combination 01, then $G_0 = G_1 = 0$, $H_0 = H_1 = 1$, where G_0, G_1, H_0, H_1 are cofactors of G and H respectively. It is obvious that $G_0 \oplus G_1 \oplus H_0 \oplus H_1 = 0$, therefore variable x has *non-skew* K4 symmetries with all the variables in Y . Similar reasoning leads to the other results in Table 3.

In case symmetries exist, the variable subset in Y is returned as potential candidates. Otherwise an empty set is returned.

Step 2. A variable z in set Y is selected and the functions are cofactored w.r.t. this variable.

Step 3. Based on Theorem 1, a variable belongs to the solution only when it belongs to the solutions of both cofactors. If one of the sub-problems returns an empty set, there is no need to solve the other one. This accounts for the efficiency the program has when the functions have no symmetries in some variables.

Table 3. Values of G/H and Corresponding Symmetries

Symmetry Type	GH= 00	GH= 01	GH= 10	GH =11
Non-skew K0	S	S		
Skew K0			S	S
Non-skew K1	S	S		
Skew K1			S	S
Non-skew K2	S		S	
Skew K2		S		S
Non-skew K3	S		S	
Skew K3		S		S
Non-skew K4	S	S	S	S
Skew K4				

Step 4. Skipped variables, as illustrated in Figure 4, need to be handled differently for different symmetry categories.

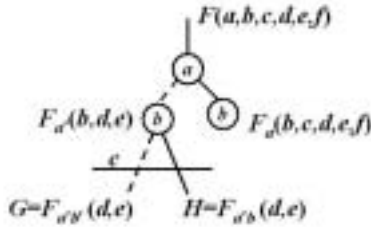


Figure 4. Illustration of skipped variables in *SymmetricVars_rec*

Constant cofactor symmetries: Skipped variables won't create additional symmetries in this case.

Classical symmetries: If G and H are equivalent, then variable x has *non-skew* T1 and T2 classical symmetries with the skipped variables. If G and H are complement of each other, then it has *skew* classical symmetries with the skipped variables.

Single variables symmetries: Regardless of the values of G and H , variable x always has *non-skew* T3 and T4 single variable symmetries with the skipped variables. If G and H are equivalent, then variable x has *non-skew* T5 and T6 single variable symmetries with the skipped variables. If G and H are complements of each other, then it has *skew* T5 and T6 single variable symmetries with the skipped variables.

Kronecker symmetries: The existence of the symmetries depend on the value of functions G and H in this case. For example, in Figure 4, variable c is not in the *support* of $F_{a'}$. The question is if b is symmetric with c . We know that $F_{a'b'c'} = F_{a'b'c} = F_{a'b'}(d, e)$ and $F_{a'b'c'} = F_{a'bc} = F_{a'b'}(d, e)$. So it is always true that variables b is symmetric with variable c with *non-skew* K4 symmetry. To check if b and c are symmetric with other types of Kronecker symmetries, for example *non-skew* K0 symmetry, we need to compute the following equation: $F_{a'b'c'} \oplus F_{a'bc'} \oplus F_{a'bc} = F_{a'b'}$. Therefore, if $F_{a'b'}$ is 0, then variable b is symmetric with variable c with *non-skew* K0 symmetry. Other symmetry conditions can be derived accordingly.

The resulting set is equal to the *union* of R_2 with the *intersection* of R_0 and R_1 . ($R = (R_0 \cap R_1) \cup R_2$)

Step 5. This step checks to see if variable x is symmetric with variable z . Cofactor relationships in Table 1 are checked

for different symmetry types. If x and z are symmetric, then z is added to the resulting set R .

4. Implementation of the algorithm

The proposed algorithm was implemented in C with the CUDD decision diagram package [18] and the EXTRA library of DD procedures [19].

Binary decision diagrams (BDDs) are used to represent Boolean functions and Zero-Suppressed Binary Decision Diagrams (ZDDs) are used to represent variable sets and symmetry graphs, as in [14]. The algorithm started by constructing shared BDDs for the function, which are not modified during the symmetry computation. No additional BDD nodes are built, which makes the implementation very fast. Similar to other BDD algorithms, partial results are cached to prevent multiple calls with the same arguments. The calls to the caching procedures are omitted in the pseudo code for simplicity. The cache lookups are performed before step 2 and the cache insertions before returning the result in both *ComputeSymmetries_rec* and *SymmetricVars_rec*.

ZDDs provide a canonical representation of the symmetry graph. The ZDDs are usually very small because the symmetry graphs are usually sparse. Although the program doesn't generate new BDD nodes, a small number of ZDD nodes are created to manipulate the symmetry graph and the variables sets in the recursive procedures. However, experiments show that the increase in the number of ZDD nodes is negligible comparing to the size of the shared BDDs of the original functions.

The worse case complexity of the algorithm is cubic in the number of the BDD nodes, because the complexity of the procedure *ComputeSymmetries_rec* is linear while each call to *SymmetricVars_rec*, performed inside *ComputeSymmetries_rec*, has the worse case quadratic complexity. However for the benchmark functions, the observed runtime is close to linear in the number of the BDD nodes.

5. Experimental results

The program was run on a 750MHz Pentium III PC running Red Hat Linux. We only compare our results with the naïve method because no other method has been proposed to compute all generalized symmetries.

MCNC benchmark function "mux" is a good example to demonstrate the efficiency of the program when there is no symmetry in the design. Table 4 shows the performance data on "mux" using both the naïve method and the new algorithm proposed in the paper. All 15 *non-skew* symmetries are checked and the results are given individually. (Times are given in seconds. The numbers reported are symmetry computation time only for all outputs of the circuit. The time it takes to read the benchmark file and construct the BDDs are not included) It can be seen that for each symmetry type, the performance speedup is one order of magnitude when there are symmetries and two orders of magnitude otherwise.

Table 5 gives the average performance gains over a number of relatively large MCNC benchmarks in two scenarios:

- Scenario 1 assumes that only one symmetry type is needed for certain application. The program computes each of the 15 *non-skew* symmetry types independently using both the naïve and fast computation method. The performance gain for each type is calculated. We derive the average gain (1) by adding the performance gains for all 15 types and then dividing the sum by 15. As can be seen, the performance improvements for individual symmetry types are very significant.
- Scenario 2 assumes that all 15 types of *non-skew* symmetries need to be computed at the same time. To achieve this, our current implementation requires running the fast computation as many times as there are symmetry types, while the naïve computation is performed by deriving the cofactors w.r.t. each pair of variables only once, and checking the relationships of these cofactors according to Table 1 to get all the symmetry information. The resulting ratio of runtimes is given in the table as average gain (2). We expect an additional speed-up of 5-10x over average gain (2) by combining all the fast computation procedures into a single BDD traversal, which is not currently done in our code. The performance improvement of the new algorithm over the naïve method in this case is very encouraging.

Table 4. Results for Benchmark *mux*

Symmetry type	Num. of symmetry	New (sec)	Naïve (sec)	Performance gain
C0	20	2.67	28.6	11
C1	0	0.11	28.54	259
C2	20	2.6	28.57	11
C3	0	0.23	28.54	124
T1	0	0.04	28.56	714
T2	0	0.04	28.54	714
T3	0	0.03	28.54	951
T4	0	0.03	28.55	952
T5	52	1.39	28.55	21
T6	32	1.37	28.54	21
K0	0	0.17	28.54	168
K1	0	0.17	28.55	168
K2	0	0.17	28.52	168
K3	0	0.18	28.52	158
K4	120	2.8	28.53	10

6. Application of generalized symmetries

Symmetry is an important characteristic of Boolean functions. Many applications in electronic design automation exploit classical symmetries to achieve better results and/or improve performance. Generalized symmetries, Kronecker symmetries in particular, are relatively new. In this section, we touch upon one natural application of generalized symmetries in

Boolean Matching. Our future work involves researching on other applications of generalized symmetries.

Table 5. Average Performance Gain on MCNC Benchmarks

Benchmarks	Num of inputs, outputs	Num of symmetries	Avg. gain (1)	Avg. gain (2)
pair	173, 137	22577	272	1.5
mux	21, 1	244	297	2.4
cm150	21, 1	204	341	4.5
c432	36, 7	212	275	13.7
c1908	33, 25	3362	210	2.3
too_large	38, 3	738	204	3.3
k2	45, 45	10104	273	8.6
c1355	41, 32	256	2017	17
c499	41, 21	256	1891	16.9
des	256, 245	15241	60	1.5
frg2	143, 139	19212	38	2.2

Boolean Matching is a technique used to determine if a subnetwork can be implemented with a given technology library during technology mapping. Two n-input Boolean functions are called NPN-equivalent if one can be transformed into the other by input permutation, input negation and output negation.

Classical symmetries have been used as signatures in Boolean Matching [5][6][25][26]. However, classical symmetries are only a small percentage of total circuit symmetries. The following lemmas and theorems state the applicability of generalized symmetries in Boolean Matching.

Lemma 1. $G(f, (i, j), g)$ implies that $G(f, (j, i), g')$, where $g' = \langle g_0, g_2, g_1, g_3, g_4 \rangle$.

Proof: $G(f, (i, j), g)$ is:

$$\forall x_1 \dots x_n [g_0 f[x_i \leftarrow 0, x_j \leftarrow 0] \oplus g_1 f[x_i \leftarrow 0, x_j \leftarrow 1] \oplus g_2 f[x_i \leftarrow 1, x_j \leftarrow 0] \oplus g_3 f[x_i \leftarrow 1, x_j \leftarrow 1] = g_4]$$

Or equivalently,

$$\forall x_1 \dots x_n [g_0 f[x_j \leftarrow 0, x_i \leftarrow 0] \oplus g_2 f[x_j \leftarrow 0, x_i \leftarrow 1] \oplus g_1 f[x_j \leftarrow 1, x_i \leftarrow 0] \oplus g_3 f[x_j \leftarrow 1, x_i \leftarrow 1] = g_4]$$

This is $G(f, (j, i), g')$, where $g' = \langle g_0, g_2, g_1, g_3, g_4 \rangle$. \square

Lemma 1 states that the symmetry types that have 1 in g_1 or g_2 are order dependent. For example, if variables x_i and x_j have *non-skew* T5 symmetry $g = \langle 1, 0, 1, 0, 0 \rangle$, then x_j and x_i will have *non-skew* T3 symmetry $g' = \langle 1, 1, 0, 0, 0 \rangle$.

Theorem 2. Permuting input variables do not alter the total number of symmetries in each category.

Proof: From Lemma 1, swapping variables does not change the symmetry category for the variable pair, because the sum of g_0, g_1, g_2, g_3 stays unchanged. Therefore, the theorem holds. \square

Definition 4. Let $\phi = \langle \phi_1, \dots, \phi_n \rangle$ denote the input phase assignments on function f . We define $(f \circ \phi)(x_1, \dots, x_n)$ to be $f(\phi_1 \oplus x_1, \dots, \phi_n \oplus x_n)$.

Lemma 2. Assume ϕ has one 1. If $\phi_i = 1$, then $G(f \circ \phi, (i, j), g) = G(f, (i, j), \langle g_2, g_3, g_0, g_1, g_4 \rangle)$; else if $\phi_j = 1$, then $G(f \circ \phi, (i, j), g) = G(f, (i, j), \langle g_1, g_0, g_3, g_2, g_4 \rangle)$; Otherwise, $G(f \circ \phi, (i, j), g) = G(f, (i, j), g)$;

Proof: Case $\phi_i = 1$, $G(f \circ \phi, (i, j), g)$ is:

$$\forall x_1 \dots x_n [g_0 \cdot (f \circ \phi)[x_i \leftarrow 0, x_j \leftarrow 0] \oplus$$

$$\begin{aligned} &g_1 \bullet (f \circ \phi) [x_i \leftarrow 0, x_j \leftarrow 1] \oplus \\ &g_2 \bullet (f \circ \phi) [x_i \leftarrow 1, x_j \leftarrow 0] \oplus \\ &g_3 \bullet (f \circ \phi) [x_i \leftarrow 1, x_j \leftarrow 1] = g_4. \end{aligned}$$

Applying the input phase assignment, we have:

$$\begin{aligned} &\forall x_1 \dots x_n [g_0 f [x_i \leftarrow 0 \oplus \phi_i, x_j \leftarrow 0 \oplus \phi_j] \oplus \\ &g_1 f [x_i \leftarrow 0 \oplus \phi_i, x_j \leftarrow 1 \oplus \phi_j] \oplus \\ &g_2 f [x_i \leftarrow 1 \oplus \phi_i, x_j \leftarrow 0 \oplus \phi_j] \oplus \\ &g_3 f [x_i \leftarrow 1 \oplus \phi_i, x_j \leftarrow 1 \oplus \phi_j] = g_4], \end{aligned}$$

which is equivalent to:

$$\begin{aligned} &\forall x_1 \dots x_n [g_2 f [x_i \leftarrow 0, x_j \leftarrow 0] \oplus g_3 f [x_i \leftarrow 0, x_j \leftarrow 1] \oplus \\ &g_0 f [x_i \leftarrow 1, x_j \leftarrow 0] \oplus g_1 f [x_i \leftarrow 1, x_j \leftarrow 1] = g_4]. \end{aligned}$$

The other two cases can be proved similarly. \square

Theorem 3. Negating input variables does not alter the total number of symmetries in each category.

Proof: From Lemma 2, input phase assignments do not change the values of g_0, g_1, g_2, g_3 , therefore the sum of g_0, g_1, g_2, g_3 stays unchanged. In case there are more than one 1 in ϕ , Lemma 2 can be applied iteratively. \square

Theorem 4. Negating output variables changes skew type for single- and three-cofactor symmetries, and preserves the skew type for two- and four-cofactor symmetries.

Proof: Let ρ be output phase assignment. $f \circ \rho = f \oplus \rho$.

$G(f \circ \rho, (i, j), g)$ is:

$$\begin{aligned} &\forall x_1 \dots x_n [g_0 \bullet (f_{00} \oplus \rho) \oplus g_1 \bullet (f_{01} \oplus \rho) \oplus \\ &g_2 \bullet (f_{10} \oplus \rho) \oplus g_3 \bullet (f_{11} \oplus \rho) = g_4], \end{aligned}$$

or equivalently,

$$\begin{aligned} &\forall x_1 \dots x_n [g_0 f_{00} \oplus g_1 f_{01} \oplus g_2 f_{10} \oplus g_3 f_{11} = \\ &g_4 \oplus ((g_0 \oplus g_1 \oplus g_2 \oplus g_3) \bullet \rho)]. \end{aligned}$$

When $\rho = 1$, for single- and three-cofactor symmetries, there are odd number of ones in g_0, g_1, g_2 and g_3 . Therefore the exclusive-or of the four is 1, giving:

$$\forall x_1 \dots x_n [g_0 f_{00} \oplus g_1 f_{01} \oplus g_2 f_{10} \oplus g_3 f_{11} = !g_4].$$

For two- and four-cofactor symmetries, the exclusive-or is 0. \square

Theorems 2, 3 and 4 state that generalized symmetries are preserved under NPN operations, and therefore can be used as signatures for Boolean matching. A simple scheme is to compute the total number of symmetries in each of the four symmetry categories, including both *skew* and *non-skew* symmetries. Since output negation changes the skew type for one and three cofactor symmetries, we use the sum of *skew* and *non-skew* symmetries in those cases. The resulting signatures are six integers representing the total number of *non-skew* and *skew* single cofactor symmetries, *non-skew* two-cofactor symmetries, *skew* two-cofactor symmetries, *non-skew* and *skew* three cofactor symmetries, *non-skew* four cofactor symmetries and *skew* four-cofactor symmetries. These signatures can uniquely classify all 2 and 3 variable functions, 124 out of the 222 NPN equivalent classes for 4 variable functions. Generalized symmetries are much more effective as signatures than using classical symmetries alone. With the efficient symmetry computation scheme proposed in this paper, generalized symmetries can be used as filters to quickly prune unnecessary tautology checks. Other techniques can also be combined with generalized symmetries to provide better results for Boolean Matching.

7. Conclusions

We presented a new algorithm to compute all pair-wise generalized symmetries for completely specified Boolean functions.

- It works on the shared BDD of multi-output functions and computes the symmetry information for each output.
- It exploits the compactness and canonicity of the ZDD representation to store the symmetry information computed for a node in the shared BDD.
- It computes all 30 types of generalized symmetries.
- It is most efficient when applied to multi-output Boolean functions with no symmetry.

Experimental results show that the overall performance of the algorithm is significantly better than the naïve symmetry computation method.

We touched upon an application of generalized symmetries in Boolean matching. We expect that the fast symmetry computation algorithm will enable and encourage more applications of generalized symmetries in electronic design automation.

References

- [1] C. R. Edwards, S. L. Hurst, "A Digital Synthesis Procedure Under Function Symmetries and Mapping Methods", *IEEE Trans. On Computers*, vol. C-27, No.11, pp. 985-997, 1978.
- [2] B.-G. Kim and D. L. Dietmeyer, "Multilevel Logic Synthesis of Symmetric Switching Functions" *IEEE Trans. CAD*, 10(4), pp.436-446, April 1991.
- [3] M. Chrzanowska-Jeske, W. Wang, J. Xia, and M. Jeske, "Disjunctive Decomposition of Switching Functions Using Symmetry Information," *Proc of IEEE SBCCI2000 International Symposium on Integrated Circuits and System Design, Manaus, Brazil*, pp. 67, September 2000.
- [4] V.N. Kravets, "Constructive Multi-level Synthesis by Way of Functional Properties", *Ph.D. Thesis*. University of Michigan, 2001.
- [5] Y.-T. Lai, S. Sastry, and M. Pedram, "Boolean Matching Using Binary Decision Diagrams with Applications to Logic Synthesis and Verification". *Proc. Intl. Conf. Computer Aided Design*, pp. 452-458, October 1992.
- [6] F. Mailhot and G. De Micheli "Technology Mapping Using Boolean Matching and Don't Care Sets" *Proc. European Design Automation Conference*, pp. 212-216, 1990.
- [7] Ch. Scholl, D. Moller, P.Molitor, and R. Drechsler, "BDD Minimization Using Symmetries". *IEEE Trans. on CAD*, 18(2) pp. 81-100, February 1999.
- [8] W. Ke, P. R. Menon, "Delay-Testable Implementation of Symmetric Functions", *IEEE Trans. on CAD*, Vol. 14, No 6 pp. 772-775, 1995.

- [9] I. Pomeranes, S. M. Reddy, "On diagnosis and correction of design errors," *Proc. ICCAD*, pp 500-507, 1993.
- [10] D.L. Cheng and M. Marek-Sadowska "Verifying equivalence of functions with unknown input correspondence" *Proc. European Design Automation Conf.* Pp. 81-85, February 1993.
- [11] D. Moller, J. Mohnke, and M. Weber, "Detection of Symmetry of Boolean Functions Represented by ROBDDs". *Proc. Intl. Conf. Computer Aided Design*, pp. 680-684, November 1993.
- [12] S. Panda, F. Somenzi and B. F. Plessier, "Symmetry detection and dynamic variable ordering of decision diagrams" *Proc. Int. Conf. Computer Aided Design*, pp. 628-631, November 1994.
- [13] C.-C. Tsai and M. Marek-Sadowska, "Generalized Reed-Muller Forms as a Tool to Detect Symmetries". *IEEE Trans. Computers*, C-45(1), pp. 33-40, January 1996.
- [14] A. Mishchenko, "Fast Computation of Symmetries in Boolean Functions", *IEEE Trans. CAD*, 22(11), pp.1588-1593, November 2003.
- [15] M. Chrzanowska-Jeske, "Generalized Symmetric and Generalized Pseudo-Symmetric Functions," *Proc. International Conference on Electronics, Circuits and Systems*, September 1999.
- [16] M. Chrzanowska-Jeske, "Generalized Symmetric Variables". *Proc. Intl. Conference on Electronics, Circuits, and Systems*. September 2001.
- [17] M. Chrzanowska-Jeske. "Regular Symmetric Arrays for Non-Symmetric Functions", *Proc. of The International Conference on Circuits and Systems*, 1999.
- [18] F. Somenzi. CUDD Package, Release 2.3.1. <http://vllis.Colorado.EDU/~fabio/CUDD/cuddIntro.html>
- [19] A. Mishchenko. EXTRA Library of DD procedures. <http://www.ee.pdx.edu/~alanmi/research/extra.htm>
- [20] R. F Arnold, M. A. Harrison, "Algebraic Properties of Symmetric and Partially Symmetric Boolean Functions," *IEEE Trans. On Electron. Computer*, Vol. EC-12, pp. 244-251, June 1963.
- [21] M. Perkowski, M. Chrzanowska-Jeske, and Y. Xu, "Lattice Diagrams Using Reed-Muller Logic", *Proc. Intl. Workshop Appl. Reed-Muller Expansions*, pp. 85-102, 1997.
- [22] P. Lindgren, R. Drechsler, B. Becker, "Synthesis of Pseudo-Kronecker Lattice Diagram", *Proc. Intl. Workshop Appl. Reed-Muller Expansions*, pp. 197-204, 1999.
- [23] A. Mishchenko, X. Wang, T. Kam. "A New Enhanced Constructive Decomposition and Mapping Algorithm". *Proc. Design Automation Conference*, pp. 143-147, June 2003.
- [24] M. Davio, J.-P. Deschamps, A. Thayse. "Discrete and Switching Functions". *McGrawHill*, 1978.
- [25] C.-C. Tsai, M. Marek-Sadowska, "Boolean Matching Using Generalized Reed-Muller Form", *Proc. Of Design Automation Conference*, June 1994.
- [26] H. Savoj, M. J. Silva, R. K. Brayton and A. Sangiovanni-Vincentelli, "Boolean Matching in Logic Synthesis", *Proc. European Design Automation Conf.*, pp. 168-174, Feb. 1992.