A Universal Symmetry Detection Algorithm

Peter M. Maurer
Dept. of Computer Science
Baylor University
Waco, Texas 76798-7356

Keywords and Phrases: Symmetric Boolean Functions, Symmetry Detection, Generalized Symmetry

ACM Categories: J.6 Computer Aided Design (CAD), B.6.4 Design Aids

General Terms: Theory, Algorithms, Design

Abstract – Research on symmetry detection focuses on identifying and detecting new types of symmetry. We present an algorithm that is capable of detecting any type of permutation-based symmetry, including many types for which there are no existing algorithms. General symmetry detection is library-based, but symmetries that can be parameterized, (i.e. total, partial, rotational, and dihedral symmetry), can be detected without using libraries. In many cases it is faster than existing techniques. Furthermore, it is simpler than most existing techniques, and can easily be incorporated into existing software. The algorithm can also be used with virtually any type of matrix-based symmetry, including conjugate symmetry.

1 Introduction

A symmetric Boolean function is a function whose inputs can be rearranged in some fashion without changing the output of the function. The importance such functions was first recognized by Shannon in [Shannon. 1949], who characterized function symmetries using permutations of the input variables. Since that time, the detection and exploitation of symmetric Boolean functions has been of recurring interest in the field of design automation [Abdollahi. 2006, Biswas. 1970, Born and Scidmore. 1968, Butler, et al. 2000, Chrzanowska-Jeske. 2001, Chung and Liu. 1998, Darga, et al. 2008, Drechsler and Becker. 1995, Hu and Marek-sadowska. 2001, Hu, et al. 2008, Ke and Menon. 1995, Kettle and King. 2008, Kravets and Sakallah. 2002, Maurer. 2011, Mohnke, et al. 2002, Moller, et al. 1993, Muzio, et al. 2008, Rice and Muzio. 2002, Scholl, et al. 1997, Tsai and Marek-Sadowska. 1996, Wang and Chen. 2004, Zhang, et al. 2004]. Virtually all of these algorithms are based on Shannon’s Theorem [Shannon. 1949] which detects symmetry by comparison of two-variable cofactors. (See Section 2 for the details.) Although comparison of two-variable cofactors is powerful enough to detect all total and partial symmetries (see Section 2 for the definitions), there are many types of symmetries that cannot be detected in this manner. As the number of input variables grows, these types of symmetry become more common than partial and total symmetry. Some progress has been made in detecting symmetries beyond partial and total symmetry [Tsai and Marek-Sadowska. 1994, Chrzanowska-Jeske. 2001, Kravets and Sakallah. 2000], but the problem of universal symmetry detection has remained open since 1949.

Our experiments with standard benchmarks [Brigle, et al. 1985] show that such symmetries are common. Other researchers have noted the existence of such symmetries [Mohnke, et al. 2002, Kravets and Sakallah. 2002]. Ignoring such symmetries can cause major failures in layout verification and regression [Maurer and Schapira. 1988]. For such algorithms, incorrect handling of symmetry can cause many false errors. When too many false errors are reported, it is easy to miss the real errors.

Correct handling of symmetry is also important when attempting to match design specifications to an existing library of functions [Mohnke, et al. 2002]. If symmetry handling fails, functions may have to be created by hand even though there is an acceptable library function to implement it. This can be costly, both in terms of time and of correctness. New implementations must be verified and tested, whereas library implementations are already verified and are much more likely to be correct.

In this paper, we use an entirely new approach which, effectively, considers all inputs simultaneously instead of in pairs. This approach allows us to detect virtually any type of symmetry, including some types that go beyond permutations. For small numbers of inputs (less than 8) our approach is faster than using cofactors. In addition, the coding is simpler. We provide pseudo code in Section 4, which can easily be adapted for use in existing EDA algorithms. Our algorithm also is somewhat easier to parallelize than the conventional algorithm, because it does not require the accumulation of results to completely characterize a function.

We cover the basic principles of symmetry in Section 2, while Section 3 focuses on Boolean orbits which are the theoretical basis of our new approach. We prove several theorems to show the correctness of this approach. Section 4 presents the universal symmetry detection algorithm. Our implementation is object-oriented with information hiding. This permits our implementation to be used with many different underlying implementations of functions and groups. Section 5 shows how to refine function orbits, should it be desirable to do so for some particular implementation. Section 6 discusses forbidden groups, and shows how to identify them. These are permutation groups (or matrix groups) that cannot be the symmetry rule of any function. Libraries are more efficient if the forbidden groups are eliminated. Sections 7 through 14 show how the Universal Symmetry Detection algorithm can be used with known types of symmetry. Because we claim that our approach is universal, it is necessary to systematically go through known types of symmetry and show that our approach can be used to detect each of these types. These sections are devoted to specialized algorithms or libraries for detecting matrix-based symmetry, partial and total symmetry, sub-symmetries, extended symmetric variable pairs, multi-phase symmetry, anti-symmetry (aka skew symmetry), Kronecker symmetry, rotational and dihedral symmetry, and auto-symmetry. Each of these algorithms and libraries have been implemented and thoroughly tested. Section 15 presents experimental results, and Section 16 draws conclusions.

2 Basic Principles

Symmetries can be categorized into total symmetry, partial symmetry, and strong symmetry. Total symmetry permits the inputs of a function to be rearranged arbitrarily without changing the output of the function. Partial symmetry is similar to total symmetry in that it permits one or more subsets of inputs to be rearranged arbitrarily.

Department of Computer Science
Baylor University
3600 University Drive
Waco, Texas 76798-7356
Strong symmetry is a catch-all term that includes every type of symmetry that is neither partial nor total. The function $x_1 + x_2 + x_3 + x_4$ is totally symmetric, the function $x_1 x_2 x_3 x_4$ is partially symmetric, while the functions $x_1' x_2 + x_3' x_4$ and $x_1 x_3 + x_2 x_4$ are strongly symmetric. (We specify functions as expressions in which multiplication signifies AND, addition signifies OR, and the prime symbol specifies NOT.) In $x_1' x_2 + x_3' x_4$, no single variable can be exchanged with any other single variable, but the set $\{x_1, x_2\}$ can be exchanged with the set $\{x_3, x_4\}$. The function $x_1 x_3 + x_2 x_4$ is more problematical because most existing algorithms will detect two partial symmetries, but ignore the fact that the set $\{x_1, x_3\}$ can be exchanged with the set $\{x_2, x_4\}$. (The algorithm of [Krats and Sakallah, 2002] will detect the correct symmetry for this function.)

There are many more kinds of strong symmetry than partial and total symmetry [Holt, 2010]. Various sub-categories of strong symmetry have been discovered, and algorithms have been created to detect and exploit some of these symmetries [Mohnke, et al. 2002]. Examples of such symmetries are hierarchical symmetry, rotational symmetry and dihedral symmetry [Krats and Sakallah, 2002].

The primary tool for categorizing symmetry is the permutation group [Passman, 1968]. Let $X$ be a finite set of objects. A permutation is a one-to-one function from $X$ to itself. In other words, a permutation rearranges the elements of $X$ without creating or destroying any elements. Permutations can be “multiplied” using function composition. If $p$ and $q$ are permutations, then so is $pq$ where $(pq)(x) = q(p(x))$. The multiplication operation is associative, $(pq)r = p(qr)$, but not necessarily commutative, $(pq)$ need not be equal $qp$. A set of permutations, $G$, that is closed under multiplication (for all $a, b \in G$, $ab \in G$) is called a Group. The set of all permutations of a set $X$ is called the symmetric group on $X$ and is written $S_X$.

Although we can apply permutations to any finite set, the only thing that affects the structure of $S_X$ is the size of $X$. If $X$ and $Y$ are two sets of the same size, then $S_X$ and $S_Y$ are identical. If $p \in S_X$, and the size of $X$ is $n$ then we say that $n$ is the degree of $p$. If $X = \{1, 2, \ldots, n\}$ we write $S_n$ as $S_n$. When speaking of the input variables of a function, we will designate the variables as $x_1, x_2, \ldots, x_n$. We will assume that all permutations are elements of $S_n$, and permute the variables $x_1, x_2, \ldots, x_n$ by operating on their indices.

Every permutation group $G$ has two important properties which are implied by $G$ being closed under multiplication. First, the identity permutation, $I$, is a member of every group. ($Ip = pl = p$ for all $p$.) Second, every permutation $p \in G$ has an inverse permutation $p^{-1} \in G$ such that $pp^{-1} = p^{-1}p = I$.

Permutations can be specified in many ways, but in this paper we will normally use cycle notation. Every permutation in $S_n$ can be characterized as one or more cyclic shifts of some subset of the integers $\{1, 2, \ldots, n\}$. For example the permutation $(1, 2, 3) \in S_3$ maps 1 to 2, 2 to 3, and 3 to 1. A permutation may perform several cyclic shifts simultaneously, as in $(1, 2, 3)(4, 5) \in S_5$, which shifts 1, 2, and 3 cyclically and also swaps 4 and 5. Elements that are not moved by the permutation are normally omitted from the cycle notation, so the permutation $(1, 2, 3)(4, 5, 6) \in S_6$ would normally be written $(1, 2, 3)$.

In a cyclic shift, it doesn’t matter which element comes first, so the cycles $(1, 2, 3)$, $(2, 3, 1)$, and $(3, 1, 2)$ all denote the same permutation. To avoid this ambiguity, the smallest cycle of a element is always listed first. A similar ambiguity occurs when a permutation performs two or more cyclic shifts of the same size, as in $(1, 2, 3)(4, 5, 6)$. In this case the cycles of the same length are written in ascending order by their smallest elements.

Cycles of length two, such as $(1, 2)$ and $(2, 3)$ are called transpositions. Any cycle of length $k$ can be factored into $k-1$ transpositions in the following manner: $(c_1, c_2, c_3, \ldots, c_k) = (c_k, c_{k-1}) (c_{k-1}, c_{k-2}) \ldots (c_3, c_2)$. This implies that any permutation can be factored into a product of transpositions. This factorization is not unique, but if a permutation, $p$, can be factored into an even number of transpositions, then there is no way to factor it into an odd number of transpositions. By the same token if a permutation can be factored into an odd number of transpositions, then there is no way to factor it into an even number of transpositions. Thus we can characterize each permutation as either even or odd. The product of two odd or two even permutations is even and the product of an odd permutation with an even permutation is odd. For every symmetric group $S_n$, there is a subgroup $A_n$ consisting of the even permutations of $S_n$. The group $A_n$ is called the alternating group of degree $n$.

Let $p$ be a permutation of degree $n$ and let $f$ be an $n$-input Boolean function. We say that $p$ and $f$ are compatible if using $p$ to rearrange the variables of $f$ leaves the output of $f$ unchanged. We also say that $f$ is invariant with respect to $p$. We extend this terminology in the obvious way to subgroups of $S_n$, and define the symmetry group $G_f$ to be the set of all permutations that leave $f$ invariant. The group $G_f$ is closed under multiplication because if $p$ and $q$ leave $f$ invariant, then so does $pq$. Thus $G_f$ is a subgroup of $S_n$. Because the identity element leaves every function invariant, $G_f$ is never empty. Most recognized types of symmetric functions can be characterized using symmetry groups. For example, an $n$-input function $f$ is totally symmetric, if and only if $G_f = S_n$. A function is non-symmetric if and only if $G_f = \{I\}$.

Most existing symmetry-detection algorithms use symmetric variable pairs, which are detected by comparing the cofactors of a function [Chrzmanowska-Jeske, 2001]. A cofactor of $f$ is found by setting one or more input variables to constant values. For example, let $f = x_1 x_2 + x_3 x_4$. Two cofactors of $f$ are $f_{\text{sub}} = x_1 x_2$ and $f_{\text{sub}} = x_3 x_4$. The four positions in the subscript correspond to the four input variables $x_1$, $x_2$, $x_3$, and $x_4$ respectively. The subscript indicates which variables have been set to constants and which are unaffected. When the unaffected variables are obvious, it is common to omit the x’s.

Symmetric variable pairs are pairs of variables that can be exchanged without affecting the output of the function. Shannon’s theorem [Shannon, 1949] states that $(x_1, x_2)$ is a symmetric variable pair if and only if $f_{0} = f_{10}$. Symmetric variable pairs are transitive, which means that if $(x_1, x_2)$ and $(x_3, x_4)$ are symmetric variable pairs, then so is $(x_1, x_3)$. Because of this, all partial and total
symmetries can be detected using symmetric variable pairs. However, symmetric variable pairs cannot be used to detect strong symmetries.

3 Boolean Orbits

The basic principle used by our algorithm is orbits. We define orbits both with respect to permutation groups and with respect to Boolean functions. This section describes how these orbits are created.

3.1 Group Orbits

Orbits have been used by mathematicians for many years to analyze and categorize permutation groups [Mohrke, et al. 2002, Passman. 1968]. They have also been used to some extent to analyze symmetric Boolean functions [Mohrke, et al. 2002]. Orbits are computed as follows. Let G be a permutation group that is compatible with a Boolean function f, and let X be the set of input variables of f. Two variables x_i, x_j ∈ X are said to be in the same orbit of G if there is a permutation p ∈ G, such that p(x_i) = x_j. Intuitively, an orbit contains all the variables that can be exchanged with one another, so the function x_1x_2x_3 + x_4 has two orbits {x_1, x_2, x_3} and {x_4}. Belonging to the same orbit is an equivalence relation, so it breaks the set of input variables into a collection of disjoint subsets.

Orbits can be used to distinguish total and partial symmetries, but are not particularly effective with strong symmetries. Consider the function x_1x_2 + x_3x_4, which possesses dihedral symmetry. At first it may appear that this function has two orbits, but in fact it has only one, {x_1, x_2, x_3, x_4}. By the same token, the totally symmetric function x_1 + x_2 + x_3 + x_4 has a single orbit, {x_1, x_2, x_3, x_4}. Thus the functions x_1x_2 + x_3x_4 and x_1x_2 + x_3x_4 have the same orbits even though their symmetries are quite different.

We have discovered a new type of orbits which we call Boolean Orbits, that permit us to deal effectively with strong symmetries as well as partial and total symmetries. Boolean orbits are computed with respect to the Boolean input vectors of a function rather than with respect to the variables. Permutations of degree n can operate on n-element vectors by permuting the indices of the elements. For example, we can apply the permutation (1,2,3) to the vector (v_1, v_2, v_3) to obtain (v_3, v_1, v_2). Applying this permutation to the specific vector (1,1,0) yields the vector (0,1,1). We formalize the concept of Boolean orbits in the following definition.

Definition 1. Given a permutation group G of degree n, two n-input vectors v and w are in the same Boolean orbit of G if there is a permutation p ∈ G such that p(v) = w.

Like ordinary orbits, belonging to the same Boolean Orbit is an equivalence relation, so this relation breaks the set of n-input Boolean vectors into a collection of disjoint sets. If G_j is the symmetry group of a Boolean function, f, then the Boolean orbits of G_j will partition the truth-table of f into disjoint sets. In fact, the symmetry of a Boolean function is completely determined by the Boolean orbits of its permutation group. Figure 1 shows the symmetry groups and the Boolean orbits of the two functions x_1 + x_2 + x_3 + x_4 and x_1x_2 + x_3x_4. The first two lines of Figure 1 give the function and the conventional orbits of the function. The final part of Figure 1 contains the Boolean orbits of the function with one Boolean orbit per line. The Boolean orbits of the two functions are quite different, even though the conventional orbits are the same. (When listing Boolean orbits, we omit all orbits of size 1, since these orbits do not affect the symmetry of the function.)

\[ x_1 + x_2 + x_3 + x_4 \]
\[ x_1x_2 + x_3x_4 \]
\[ \{x_1, x_2, x_3, x_4\} \]
\[ \{x_1, x_2, x_3, x_4\} \]

0001 0010 0100 1000
0111 0110 1001 1010 1110
0110 1101 1110 1001 0110
0111 1011 1100 1101 0100
0011 0100 1000 0110 1001

Figure 1. Orbits, Symmetry Groups, and Boolean Orbits.

We can summarize the important properties of Boolean orbits in the following theorems. Theorem 1 states that a symmetric Boolean function must map the elements of each of the Boolean orbits of its symmetry group to a unique value.

Theorem 1. Let f be a Boolean function, and G_j be the symmetry group of f. If K is a Boolean orbit of G_j and u,v ∈ K, then \( f(u) = f(v) \).

Proof. If K is a Boolean orbit of G_j, and u,v ∈ K, then there is a permutation p ∈ G_j such that p(u) = v. Since every element of G_j must leave f invariant, \( f(u) = f(p(u)) = f(v) \).

Theorem 2 is the converse of Theorem 1. It states that if the Boolean function, f, maps the orbits of a symmetry group, G, to unique values, then f is compatible with G.

Theorem 2. Let f be an n-input Boolean function and let G ⊆ S_n be a group such that for every Boolean orbit, K, of G, and for every pair of elements u,v ∈ K, \( f(u) = f(v) \) then f is invariant with respect to G, and G ⊆ G_j.

Proof. Let p be an element of G, and let u be any input of f. The vectors u and p(u) are in the same Boolean orbit, K, of G. Since \( f(u) = f(v) \) whenever u,v ∈ K, \( f(u) = f(p(u)) \), and f is invariant with respect to p. Since p was arbitrary, every element of G is compatible with f. The group G_j contains every permutation that leaves f invariant, so if p ∈ G then p ∈ G_j, and G ⊆ G_j.

Obviously, the singleton orbits (those containing a single vector) do not affect the symmetry of a function, so when testing a function f for compatibility with a group G, the singleton orbits can be ignored.

Theorem 3 deals with the problem of functions that have more than one type of symmetry. As this theorem shows, if a Boolean function f has two different types of symmetry A and B, then f also possesses an overarching symmetry that includes both A and B. Thus if we are able to identify the largest symmetry group that is compatible with f, then we are guaranteed to have discovered all of the symmetries possessed by f.
Theorem 3. Let \( f \) be a Boolean function and let \( G \) and \( H \) be two permutation groups that are compatible with \( f \). Then there is a permutation group, \( K \) compatible with \( f \) such that and \( G \) and \( H \) are both subgroups of \( K \).

Proof. Let \( K \) be the smallest subgroup of \( S_n \) containing \( G \cup H \). Since \( S_n \) contains both \( G \) and \( H \), \( K \) must exist. From group theory, we know that every element of \( p \in K \) is of the form \( p = q_1q_2 \ldots q_n \) where \( q_i \in G \) or \( q_i \in H \). Since every element of either \( G \) or \( H \) is compatible with \( f \), \( p \) must also be compatible with \( f \), and \( K \) is the required group.

In the remainder of the paper we will make extensive use of the characteristic function of an orbit. Let \( S \) be any set of \( n \)-element Boolean vectors. The characteristic function of \( S \), \( C \), is an \( n \)-input Boolean function which is equal to 1 on every element of \( S \), and zero elsewhere. Figure 2 gives a set of orbits along with their characteristic functions in truth-table form.

![Figure 2. Boolean Orbits and Characteristic Functions.](image)

The characteristic functions of the Boolean Orbits can be used to determine the symmetries of a Boolean function. We expand on this idea in the following section.

Given a permutation group, \( G \), computing the Boolean orbits of \( G \) is straightforward. The algorithm is given in Figure 3.

Set Orbit-Collection \( B \) to Empty
For Each \( n \)-element Boolean vector \( v \)
If \( v \) has not been assigned to an orbit
Create a new orbit \( Z \)
Assign \( v \) to \( Z \)
Mark \( v \) as assigned to an orbit
For Each permutation \( p \) in group \( G \)
Apply \( p \) to \( v \) to obtain \( w \)
If \( w \) is not in \( Z \)
Assign \( w \) to \( Z \)
Mark \( w \) as assigned to an orbit
End If
End For
End If
End For

Figure 3. Generating Boolean Orbits.

Shannon’s theorem, which is the basis of virtually all other symmetry detection algorithms, is a special case of Theorem 2. Shannon’s theorem deals with symmetric variable pairs of a function \( f \), \( (x_i, x_j) \). Formally, we define symmetric variable pairs as follows.

Definition 2. Let \( f \) be an \( n \)-input Boolean function and let \( x_i \) and \( x_j \) be two input variables of \( f \). The pair \( (x_i, x_j) \) is a symmetric variable pair if and only if \( f \) is compatible with the permutation group \( \{I, (i, j)\} \).

Theorem 4 shows that Shannon’s theorem is a special case of Theorem 2.

Theorem 4. Let \( f \) be a Boolean function, and let \( f_{i0}, f_{i1} \) be the cofactors of \( f \) with respect to the pair of variables \( (x_i, x_j) \). The pair \( (x_i, x_j) \) is a symmetric variable pair of \( f \), if and only if \( f_{i0} = f_{i1} \).

Proof. If \( (x_i, x_j) \) is a symmetric variable pair of \( f \), then \( f \) must be invariant with respect to the permutation group \( \{I, (i, j)\} \). The Boolean orbits of this group are either singletons of the form \( \{\ldots 0 \ldots 0 \ldots \}, \{\ldots 1 \ldots 1 \ldots \} \), or \( \{\ldots 0 \ldots 1 \ldots \ldots \} \). Now consider two vectors of the form \( \ldots 0 \ldots 1 \ldots \ldots \) and \( \ldots 1 \ldots 0 \ldots \ldots \). If combine all vectors of the form \( \ldots 0 \ldots 1 \ldots \ldots \) into a single set, and the 0, and the 1, we obtain the truth table of the cofactor \( f_{i0} \). We obtain truth table of the cofactor \( f_{i1} \) in the same manner. Because each element of each Boolean orbit must be mapped to a single value, the truth tables must be identical.

3.2 Function Orbits

For an \( n \)-input Boolean function, \( f \), we divide the set \( n \)-element Boolean vectors into two sets \( U_f \) and \( Z_f \), which are the points where \( f \) takes the value one and the value zero respectively. The sets \( U_f \) and \( Z_f \) are called the Boolean orbits of \( f \). The characteristic functions of these sets are just \( f \) and \( \overline{f} \). (If \( f(v) = 0 \) then \( \overline{f}(v) = 1 \) and if \( f(v) = 1 \) then \( \overline{f}(v) = 0 \).)

Let \( f \) and \( g \) be two \( n \)-input Boolean functions. The function \( f \) is said to imply \( g \) if \( g(v) = 1 \) whenever \( f(v) = 1 \). We can use the concept of implication to define the fundamental relationship on which our symmetry detection algorithm is based. This result is given in Theorem 5.

Theorem 5. Let \( G \subseteq S_n \) be a permutation group, and let \( f \) be an \( n \)-input Boolean function. Let \( K = \{K_1, K_2, \ldots, K_k\} \) be the collection of Boolean orbits of \( G \) with characteristic functions \( \{C_1, C_2, \ldots, C_k\} \). The group \( G \) leaves \( f \) invariant if and only if \( C_i \) implies \( f \) or \( \overline{f} \) for every \( i, 1 \leq i \leq k \).

Proof. Suppose \( C_i \) implies \( f \), and \( v \in K_i \). Then \( C_i(v) = 1 \), and because \( C_i \) implies \( f \), \( f(v) = 1 \) for all \( v \in K_i \). Now suppose \( C_i \) implies \( \overline{f} \). If \( v \in K_i \) then \( C_i(v) = 1 \) and since \( C_i \) implies \( \overline{f} \), \( \overline{f}(v) = 1 \), implying that \( f(v) = 0 \) for all \( v \in K_i \). By Theorem 2, \( f \) must be invariant with respect to \( G \).
Now suppose \( f \) is invariant with respect to \( G \). By Theorem 1, if \( u, v \in K_i \), then \( f(u) = f(v) \). But \( C_i(u) = C_i(v) = 1 \). If \( f(u) = f(v) = 1 \), then \( C_i \) implies \( f \). If \( f(u) = f(v) = 0 \), then \( C_i \) implies \( \bar{f} \).

Theorem 5 gives us a principle that we can use to detect symmetry with respect to any permutation group. Given a permutation group \( G \) it is straightforward to compute the characteristic functions of its Boolean orbits. Given \( f \) it is straightforward to compute \( \bar{f} \). Once these functions have been computed, we only need to check the characteristic function of each orbit to determine whether \( f \) is symmetric with respect to \( G \).

Although most symmetry detection is normally done on single-output functions, the principle can be extended easily to multiple-output functions. First we need to define the orbits of multiple output functions.

**Definition 3.** Let \( f \) be a Boolean function with \( n \) inputs and \( m \) outputs. For each \( m \)-element vector \( v \), \( S_f(v) \) is the set of all \( n \)-element input vectors on which \( f \) takes the value \( v \). The sets \( S_f(v) \) are known as the Boolean orbits of \( f \). The function \( f(v) \) is the characteristic function of \( S_f(v) \).

Theorem 6 extends the implication principle to Boolean functions with multiple outputs.

**Theorem 6.** Let \( G \subseteq S_n \) be a permutation group, and let \( f \) be an \( n \)-input Boolean function with \( m \) outputs. Let \( K = \{K_1, K_2, \ldots, K_k\} \) be the collection of Boolean orbits of \( G \) with characteristic functions \( \{C_1, C_2, \ldots, C_k\} \). The group \( G \) leaves \( f \) invariant if and only if for every \( i, 1 \leq i \leq k \), \( C_i \) implies \( f(v) \) for some \( m \)-element vector \( v \).

The proof is essentially identical to that of Theorem 5.

**4 The Symmetry Detection Algorithm**

Figure 4 gives the pseudo code for the Universal Symmetry Detection (USD) algorithm. In this figure, it is assumed that the algorithm is being applied to a collection of functions, and that a library of symmetries is being used. Each library entry contains a set of characteristic functions that correspond to the Boolean orbits of a symmetry group. In most cases, the library will contain all subgroups of \( S_n \) for some integer, \( n \), although we have a number of other more specialized libraries. We currently have complete libraries for \( S_n \) through \( S_9 \). Subgroup libraries for \( S_9 \) through \( S_{18} \) exist on the web [Pfeiffer, 2005, Holt, 2010], but we have not yet adapted these libraries for use with the USD. When used with a complete library for \( S_n \), symmetry detection begins with the largest group so the algorithm may stop as soon as a compatible group is found.

As Figure 4 shows, the algorithm reads each function, and compares it to each library entry until a compatible entry is found. Most libraries contain a “non-symmetric” entry which permits each function to be associated with at least one library entry. However, such an entry is not required. If no compatible subgroup can be found, the function is marked as non-symmetric.

Comparison between a function and a subgroup is done by enumerating the Boolean orbits of the subgroup. Each Boolean orbit is tested against \( f \) and \( \bar{f} \) seeking an implication. If a particular subgroup orbit does not imply either function orbit, the comparison with the group is terminated. However, if all subgroup orbits imply a function orbit, the subgroup is assigned to the function as its symmetry group, and testing of the function terminates.

Libraries are not necessary for symmetries that can be parameterized for an arbitrary number of inputs. As yet, only a few symmetries have been so categorized, the most well-known of which are total, symmetry, partial symmetry, rotational symmetry, dihedral symmetry, and various types of hierarchical symmetry. The USD algorithm has special generators for total, partial, dihedral and rotational symmetry, which permits these types of symmetries to be detected without having a precomputed library. Of course it is possible to cache the output of these generators for future use.

**Load Library**

Sort Library into descending order by subgroup size.

For each function \( f \) of \[ f = \{C_1, C_2, \ldots, C_k\} \] do:

For each subgroup \( G \) in Library:

If \( G \) is compatible with \( f \) do:

For each orbit \( K \) of \( G \) while GroupCompatible = True do:

If \( f \) is compatible with \( K \) do:

Assign \( G \) as the symmetry group of \( f \);

EndIf

EndFor

If Not OrbitCompatible do:

Mark \( f \) as nonsymmetric

EndIf;

EndFor;

EndIf;

EndFor;

End

**Figure 4.** The Universal Symmetry Detection Algorithm.

Most of our libraries contain one entry per symmetry group. However, for large numbers of inputs it is not feasible to store libraries in this fashion. (See Figure 5.) We use the conjugacy relation to reduce the size of the library for large numbers of inputs. Certain types of symmetry are fundamentally the same, but applied to different inputs, and certain types of symmetry are fundamentally different. For example, a 3-input partial symmetry on the first three inputs of a function is not fundamentally different from a three-input partial symmetry on the last three variables. But a partial symmetry in the first three variables is fundamentally different from a partial symmetry in the first two variables. The conjugacy relation is used to distinguish symmetries that are essentially the same from symmetries that are fundamentally different. Definition 4 allows us to formalize this idea.

**Definition 4.** Two permutations \( p \) and \( q \) are conjugate to one another if there is another permutation \( s \) such that \( p = s^{-1}qs \).

Conjugacy can be best understood by visualizing it in this way: to permute the last three variables of a function, we move them to the
first three variables using $s$, then apply $q$ to the first three variables, and then use $s^{-1}$ to move the variables back where they were.

This relationship can be extended to permutation groups in the following way: $s^{-1}Gs = \{s^{-1}ps \mid p \in G\}$. If two symmetries are fundamentally the same then their permutation groups will be conjugate to one another. For example, all partial symmetries on three inputs have conjugate symmetry groups.

Conjugacy is an equivalence relation, so the subgroups of a group can be partitioned into a set of conjugacy classes. Figure 5 shows the number of conjugacy classes and the number of subgroups for the symmetric groups from $S_1$ through $S_{18}$. In the full libraries, we store each subgroup of the symmetric group. In reduced libraries we store only one member of each conjugacy class.

| Degree | Classes | Subgroups |
|--------|---------|-----------|
| 1      | 1       | 1         |
| 2      | 2       | 2         |
| 3      | 4       | 6         |
| 4      | 11      | 30        |
| 5      | 19      | 156       |
| 6      | 56      | 1455      |
| 7      | 96      | 11,300    |
| 8      | 296     | 151,221   |
| 9      | 554     | 1,694,723 |
| 10     | 1593    | 29,594,446|
| 11     | 3094    | 404,126,228|
| 12     | 10,723  | 10,594,925,360|
| 13     | 20,832  | 175,238,308,453|
| 14     | 75,154  | 5,651,774,693,595|
| 15     | 159,129 | 117,053,117,995,400|
| 16     | 686,165 | 5,320,744,503,742,316|
| 17     | 1,466,358 | 125,889,331,236,297,288|
| 18     | 7,274,651 | 7,598,016,157,515,302,757|

Figure 5. Subgroups and Conjugacy Classes of $S_n$ [Holt. 2010].

To regenerate a conjugacy class, it is necessary to compute the conjugates of each library entry. However for each subgroup $G$ of $S_n$, there are many pairs of permutations $(p, q)$ such that $p \neq q$, but $p^{-1}Gp = q^{-1}Gq$. To avoid duplicated work we store a set of permutations with the library entry. There is one permutation for each conjugate permutation to the entry will restore the entire class.

The permutations are computed when creating the library. A group theoretic result states that “the number of conjugates of a group is equal to the index of its normalizer [Robinson. 1995].” The normalizer of a group is the set of permutations that leave $G$ unchanged with respect to conjugacy. That is, the set $\{p \mid p^{-1}Gp = G\}$. If $G$ is a subgroup of $S_n$, then its index is equal to $[S_n]/[G]$, which is the number of right cosets of $G$ in $S_n$. A right coset of $G$ is obtained by multiplying every element of $G$ by some element $p$ of $S_n$. It is written $Gp$.

Let $N(G)$ be the normalizer of $G$. If $p$ and $q$ are members of the same right coset of $N(G)$, then $p^{-1}Gp = q^{-1}Gq$. If the permutations come from two different right cosets, then the conjugates will be different. A set of coset representatives, which includes one permutation from each right coset of $N(G)$, can be used to generate the entire conjugacy class of $G$. Figure 6 gives the algorithm for generating a reduced library entry. Figure 7 gives the library entry that

is used to detect 2-variable partial symmetry in 3-input functions. The permutations are coded in the form of a list of numbers from the set $\{0,1,2\}$. The first permutation is the identity, $I$.

The first line of Figure 7 is the name of the symmetry, the second is the number of inputs, the third is the list of coset representatives, and the remainder is the Boolean orbits, one orbit per line.

For the larger symmetric groups, reconstructing all conjugacy classes is a physical impossibility. So instead we use the set of stored permutations to alter the function under test. Let’s suppose that $g$ is invariant with respect to $p^{-1}Gp$. Then, there is an $f$ which is invariant with respect to $G$ such that $p^{-1}f = g$. If $g$ is invariant with respect to $p^{-1}Gp$, then $pg$ is invariant with respect to $G$. Figure 8 gives the pseudocode for detecting symmetry with reduced library entries. The test for compatibility in Figure 8 is identical to that in Figure 4. In most cases, this code will be slower than using a fully expanded library, so the algorithm of Figure 8 is used only when necessary.

```
Read group G
Normalizer = {};
For each permutation $p$ in $S_n$
    If $p^{-1}Gp = G$
        Add $p$ to Normalizer
    Endif
EndFor
CosetCollection = {};
For each permutation $p$ in $S_n$
    If $Gp$ is not in CosetCollection
        Add $Gp$ to CosetCollection
    EndIf
EndFor
CosetRepresentatives = {};
For each Coset $c$ in CosetCollection
    Select one element $p$ from $c$;
    Add $p$ to CosetRepresentatives
EndFor
Write G with CosetRepresentatives
Figure 6. Generating Coset Representatives.
```

Figure 7. A Reduced Library Entry.
Read function \( f \);

\[ \text{FoundSymmetry} = \text{False}; \]

For each group \( G \) while Not FoundSymmetry

For each coset representative \( p \) in \( G \)

Compute \( pf \)

if \( pf \) is compatible with \( G \)

Assign \( p^{-1}gp \) as the symmetry group of \( G \)

\[ \text{FoundSymmetry} = \text{True}; \]

Break;

EndFor

Figure 8. The Reduced-Library Detection Algorithm

5 Refining Function Orbits

Boolean orbits have two important properties that can be useful in some applications. The first concerns the weight, \( w(v) \), of a vector \( v \), which is the number of ones in the vector.

Theorem 6. Let \( K \subseteq B^n \) be a Boolean orbit of some group \( G \subseteq S_n \), and let \( u,v \in K \). Then \( w(u) = w(v) \).

Proof. If \( u,v \in K \), then there exists a \( p \in G \) such that \( p(u) = v \).

Since \( p \) rearranges the elements of \( u \) without changing them, no ones can be added or deleted. Therefore \( u \) and \( v \) must have the same number of ones, and \( w(u) = w(v) \).

Theorem 6 implies that the minimum number of orbits for any \( n \)-input function is \( n+1 \). The next property involves the complement \( v' \) of a vector \( v \). Let \( v = (v_1,v_2,...,v_n) \) and \( v' = (v_1',v_2',...,v_n') \). If \( v_i = 1 \), then \( v_i' = 0 \) and vice-versa. We start with the following lemma.

Lemma 1. Suppose \( v \) is an \( n \)-element Boolean vector, and that \( p \) is a permutation of order \( n \). If \( p(v) = u \), then \( p(v') = u' \).

Proof. Let \( u = (u_1,u_2,...,u_n) \), and \( v = (v_1,v_2,...,v_n) \).

Because \( p(v) = u \), \( u_i = v_{p(i)} \). And \( u = p(v) = (v_{p(1)},v_{p(2)},...,v_{p(n)}) \).

But by the same token, \( p(v') = (v'_{p(1)},v'_{p(2)},...,v'_{p(n)}) = u' \).

Let \( S \) be a set of \( n \)-element Boolean vectors. Then \( S' = \{v' \mid v \in S \} \). We can apply Lemma 1 to Boolean orbits to obtain the following result.

Theorem 7. If \( K \) is a Boolean orbit of a group \( G \subseteq S_n \), then \( K' \) is also a Boolean orbit of \( G \).

Proof follows immediately from Lemma 1.

It is possible to refine the Boolean orbits of a function using Theorem 6. This may be advantageous for some implementations of characteristic functions.

Suppose we have a Boolean function \( f \), with Boolean orbits \( U_f \) and \( Z_f \), and suppose we have a group \( G \) with orbits \( \{K_1,K_2,...,K_s\} \). Because of Theorem 6, we know that every vector contained in \( K_i \) must have the same weight, \( i \), we need only concern ourselves with the weight-\( i \) elements of \( U_f \) and \( Z_f \) when testing for implication. Thus, we can break \( U_f \) and \( Z_f \) into smaller sets based on weight, \( \{U_{f,0},U_{f,1},...,U_{f,i}\} \) and \( \{Z_{f,0},Z_{f,1},...,Z_{f,i}\} \), where all elements of \( U_{f,i} \) or \( Z_{f,i} \) have weight \( i \).

The Boolean orbits of a function can be further refined using Theorem 7. Suppose two vectors \( v \) and \( w \) are in the same Boolean orbit of \( f \), but \( v' \) and \( w' \) are in different orbits. From Theorem 7 we know that if \( v' \) and \( w' \) are in different Boolean orbits of a group, \( G \), then \( v \) and \( w \) must be in different Boolean orbits as well. When testing for implication, we need concern ourselves with at most one of \( v \) and \( w \). Therefore, we can place \( v \) and \( w \) in different function orbits, refining the orbit containing both \( v \) and \( w \).

The refinement procedure works as follows. Let \( O_i \) be a function orbit such that \( O_i' \) is not a function orbit, and let \( O_2 \) be a function orbit such that \( O_2 \cap O_i' \neq \emptyset \). We replace \( O_i \) with \( (O_2 \cap O_i') \) and \( (O_i' - O_2) \) and we replace \( O_2 \) with \( O_2 \cap O_i' \) and \( O_2 - O_i' \). This process continues until no further refinement is possible.

Function Orbit refinement is not an essential part of the universal symmetry detection algorithm. Depending on the underlying implementation, it may be advantageous to have a few large orbits as opposed to many small orbits. In our implementation we skip function orbit refinement, but for many implementations it may prove advantageous.

6 Forbidden Groups

There are certain permutation groups, and certain matrix groups that cannot be the symmetry group of any function. For example, suppose \( G \) is a permutation group with Boolean orbits \( B \), and suppose there is another group \( H \) such that \( G \subseteq H \subseteq \mathbb{Z} \) such that \( H \) has also has Boolean orbits \( B \). In this case, \( G \) cannot be the symmetry group of any function, because any function, \( f \) that is compatible with \( G \) will also be compatible with \( H \), but \( G \) does not contain every permutation that is compatible with \( f \), so \( G \) cannot be the symmetry group of \( f \).

We call groups of this nature forbidden groups. Forbidden groups can arise in two different ways. Some forbidden groups are output limited and some are input limited. Input limited groups arise when two different groups have the same Boolean orbits. Output limited groups are more complicated. For example, the group \( K4 = \{1, (1,2),(3,4), (1,3)(2,4), (1,4)(2,3)\} \) is output limited. (This group is known as the Klein 4-group, hence the name K4.) The Boolean orbits of K4 are given in Figure 9.

\[
\begin{align*}
0001 & 0010 0100 1000 \\
0011 & 1100 \\
0101 & 1010 \\
0110 & 1001 \\
0111 & 1011 1101 1110
\end{align*}
\]

Figure 9. The Boolean Orbits of K4.

These orbits are quite similar to the Boolean orbits of the three conjugates of \( D_4 \), the dihedral group of order 8. These orbits are given in Figure 10.

\[
\begin{align*}
0001 & 0010 0100 1000 \\
0011 & 1100 \\
0101 & 1010 \\
0110 & 1001 \\
0111 & 1011 1101 1110
\end{align*}
\]

Figure 10. The Boolean Orbits of D4.
The problem with K4 lies with the three orbits \{0011,1100\}, \{0101,1010\}, and \{0110,1010\}. If a function maps the first two orbits to the same value, the result will be a dihedral symmetry of type D8.2. If it maps the last two orbits to the same value, the result will be a dihedral symmetry of type D8.3. If it maps the first and last orbits to the same value, the result will be dihedral symmetry of type D8.1.

Lemma 2. Let \( n \) be greater than 2, \( v \) be an \( n \)-element vector, and \( p \) be a permutation of degree \( n \). If \( p(v) = w \), then there is an even permutation \( q \) such that \( q(v) = w \).

Proof. Suppose \( p \) is odd. Because \( n \) is greater than 2, there must be two elements of \( v \) that are identical. Suppose these are positions \( i \) and \( j \), with \( i \neq j \). Thus \((i,j)\) is a 2-cycle, and \( q = (i,j)p \) is an even permutation. But \((i,j)(v) = v \), and \( q(v) = p((i,j)(v)) = p(v) = w \).

Figure 11. A Function with K4 Symmetry.

\[
S_3 = \{I,(1,2),(1,3),(2,3),(1,2,3),(1,3,2)\}
\]

\[
A_3 = \{I,(1,2,3),(1,3,2)\}
\]

001 010 100
011 101 110
Figure 12. The Boolean Orbits of \( S_3 \) and \( A_3 \).

Theorem 8. For all \( n > 2 \), \( A_3 \) is forbidden.

Proof. Suppose \( u \) and \( v \) are in the same Boolean orbit of \( S_3 \). Then there must be a \( p \in S_3 \) such that \( p(u) = v \). By Lemma 2, there must be a \( q \in A_3 \) such that \( q(u) = v \). Therefore \( u \) and \( v \) must be in the same Boolean orbit of \( A_3 \). Thus every Boolean orbit of \( S_3 \) must be contained in a Boolean orbit of \( A_3 \). Because the Boolean orbits of \( S_3 \) are as large as possible, the Boolean orbits of \( S_3 \) and \( A_3 \) must be identical, and because \( A_3 \subseteq S_3 \), \( A_3 \) must be forbidden.

The groups \( A_3 \) are examples of input limited groups. They are forbidden because there are only two distinct values for each input. If we were to consider functions with multiple-valued inputs, not all of the alternating groups would be forbidden. (Extended versions of Lemma 2 and Theorem 8, would still be true for sufficiently large \( n \).)

As an example, the subgroup of \( S_6 \), \( S3a = \{I,(1,2,3)(4,5,6),(1,3,2)(4,6,5)\} \), is isomorphic to \( A_4 \) (but not conjugate to \( A_4 \)). \( S3a \) is not forbidden, in fact the function \( x_0 x_1 x_2 x'_3 + x_2 x_3 x'_6 + x_3 x'_5 x_6 \) possesses \( S3a \) symmetry. This function is derived from the three cubes of Figure 13, which make it easy to see that a rotation of the first three variables must be accompanied by a rotation of the last three variables. The reason that \( S3a \) is not forbidden is that the inputs operate in pairs: \( \{x_0, x_1\}, \{x_2, x_3\} \) and \( \{x_5, x_6\} \). Each pair of inputs has four possible values, so the argument of Theorem 8 does not apply.

\[
\begin{align*}
xx10x1 \\
x1xx10 \\
1xx10x
\end{align*}
\]

Figure 13. The cubes of \( x_0 x_1 x'_2 + x_2 x_3 x'_6 + x_3 x'_5 x_6 \).

The elimination of forbidden groups from symmetry libraries is an important optimization step, because it eliminates symmetry tests that can never succeed.

Matrix-based symmetry also displays the phenomenon of forbidden groups.

7 Sub-Symmetries

For functions with many inputs, it may be more useful to detect smaller, more manageable symmetries on a subset of inputs. We call such symmetries Sub-Symmetries. The USD algorithm is capable of detecting sub-symmetries using two different techniques.

The first technique is to “promote” existing symmetry rules to a collection of rules for a larger number of inputs. Although this process is technically feasible, it is cumbersome, and can be extremely slow for reduced library entries.

The second procedure alters the functions under test rather than the libraries. It is based on the following Theorem 9.

Theorem 9. Let \( R \) be a symmetry group of degree \( k \), and let \( f \) be a function of \( n > k \) inputs. Let \( S \) be a subset of \( k \) inputs taken from the \( n \) inputs of \( f \). If \( f \) possesses \( R \) symmetry in the set of \( k \) variables, then every cofactor obtained by fixing the \( n-k \) variables to constant values must possess \( R \) symmetry.

The Proof is obvious.

There are \( 2^{n-k} \) such cofactors for each set of \( k \) inputs. When testing an \( n \)-input function using a symmetry rule of degree \( k < n \),
the USD algorithm begins by generating all combinations of \( n \) inputs taken \( k \) at a time. For each combination, the USD algorithm generates all \( 2^{n-k} \) cofactors, and tests each one for \( R \) symmetry. Figure 14 gives the algorithm for detecting sub-symmetries. The algorithm for generating combinations can be found in the next section (Figure 17). It generates an ascending sequence of \( n-k \) numbers taken from the set \( \{0,1,\ldots,n-1\} \).

Read the Boolean Orbits of \( R \)
For each combination \( C \) of \( n \) things taken \( n-k \) at a time
  For each \( n-k \)-element vector \( V \)
    // Obtain the cofactor of \( f \)
    \( E = \) Set the variables of \( f \) specified by \( C \) to the values of \( V \)
    If \( R \) is not compatible with \( E \)
      Return “No Sub-Symmetry”
  EndIf
EndFor
Return \( C-Bar \)
Figure 14. The Sub-Symmetry Detection Algorithm.

The algorithm then tests for a sub-symmetry in the \( k \) variables selected by the combination. Each such test requires computing \( 2^{n-k} \) cofactors. These cofactors are computed by setting the variables not selected by the permutation to every possible combination of zeros and ones. The procedure continues until a sub-symmetry is found, or until all combinations have been exhausted. The \( C-Bar \) return value is a combination of \( n \) things taken \( k \) at a time that specifies the variables containing the sub-symmetry. It is easily computed from \( C \).

The algorithm of Figure 14 can be easily modified to find multiple sub-symmetries of the same type, however to avoid detecting overlapping symmetries, it is better to run the algorithm of Figure 14 on cofactors of \( f \) that do not include the variables specified by \( C-Bar \).

8 Partial Symmetries

In some cases, it may be desirable to restrict the focus to partial and total symmetries. These symmetries are well understood, and can be exploited by several existing algorithms. The Universal Symmetry Detection algorithm can be used to detect these types of symmetries without testing each pair of variables. We have provided a generator for detecting these types of symmetries.

The partial/total symmetry generator algorithm is one of our most complex algorithms. It is based on the partitions of a integer, \( n \). A partition of a positive integer \( n \) is a sequence of positive integers, \( k_1,k_2,\ldots,k_q \), such that for all \( 0 < i < q \), \( k_i \leq k_{i+1} \) and \( \sum_{i=1}^{q} k_i = n \). Figure 15 lists the partitions of 5, 6, and 7.

A partition such as 4,3,2,1 represents a 10-input function, \( f \), with a partial symmetry in four of the inputs, another partial symmetry in three different inputs, and a final partial symmetry in two inputs. The final input is not symmetric with any other input. By conjugation, we can move the four distinct groups anywhere in the input list. In any partition, a 1 represents an input that is not symmetric with any other, while the numbers greater than one represents a group of inputs that are symmetric with one another. The algorithm for computing the partitions of a number \( n \) is fairly straightforward and is given in Figure 16. For simplicity, we assume that each partition consists of \( n \) integers, some of which may be zero.

Each partition represents a large number of different symmetries. For example, the partition 3,1 represents a four-input function with a partial symmetry in three variables. There are four distinct partial symmetries of this type, as exemplified by the four functions \( x_1x_2x_3 + x_4 \), \( x_1x_2x_4 + x_3 \), \( x_1x_3x_4 + x_2 \), and \( x_2x_3x_4 + x_1 \). We must generate a symmetry rule for each of these partial symmetries.

When a partition contains a single number greater than one, as in 4,1 and 2,1,1,1, finding the distinct symmetry types is straightforward. We have created a new data type called “Combination” that represents a combination of \( n \) things taken \( m \) at a time. We use the combination as an index into the array of variable names to obtain a set of variables.

In practical terms, a Combination is an array of \( m \) numbers taken from the set \( \{0,1,2,\ldots,n-1\} \). The numbers in the array must appear in ascending sequence, and there may not be any duplicates. Although the algorithm for generating successive combinations is well known, it is not familiar to everyone, so we include it here, in Figure 17, for completeness.
situation is even more complicated. Consider the partition of 5: 2,2,1.

If we blindly follow the method given above, we will generate duplicate partial symmetry types. This occurs because the grouping of inputs \((x_1,x_3)(x_2,x_4)x_5\) is identical to the grouping \((x_1,x_4)(x_2,x_3)x_5\). When generating sets of inputs of the same size, it is necessary to guarantee that the first elements of the identically sized sets occur in ascending order. When computing the number of partial symmetries generated by a partition, we must first identify all sequences of repeating digits greater than 1, and divide the total by \(k!\), where \(k\) is the length of the repeating sequence. For example, consider the partition of 16, 3,3,3,2,2,1,1,1. The number of partial symmetries defined by this partition is given by the following formula.

\[
\frac{\begin{pmatrix} 16 \\ 3 \\ 3 \\ 3 \\ 2 \\ 2 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}}{3!} = \frac{560 \times 286 \times 120 \times 21 \times 10}{6 \times 2} = 336,336,000
\]

Figure 18 gives the total number of partial symmetries for each number of inputs from 1 through 18.

| Degree | Partitions | Partial/Total Symmetries |
|--------|------------|--------------------------|
| 1      | 1          | 1                        |
| 2      | 2          | 2                        |
| 3      | 3          | 5                        |
| 4      | 5          | 15                       |
| 5      | 7          | 52                       |
| 6      | 11         | 203                      |
| 7      | 15         | 877                      |
| 8      | 22         | 4,140                    |
| 9      | 30         | 21,147                   |
| 10     | 42         | 115,975                  |
| 11     | 56         | 687,570                  |
| 12     | 77         | 4,213,579                |
| 13     | 101        | 27,644,437               |
| 14     | 135        | 190,899,322              |
| 15     | 176        | 1,382,958,545            |
| 16     | 231        | 10,480,142,147           |
| 17     | 297        | 82,864,869,804           |
| 18     | 385        | 682,076,806,159          |

Figure 18. Total and Partial Symmetries in \(S_n\).

Once a partition has been generated, a set of symmetry rules must be created for it. In each partition, we focus on the numbers greater than one. Each such number designates a set of mutually symmetric inputs, however this set of inputs can be distributed in arbitrary fashion throughout the input list of the function. For each number greater than one, we allocate a data structure identical to that of Figure 19. These data structures are placed in an array in the order that the corresponding numbers appear in the partition. Each data structure has three sets of inputs. Available Inputs list is the collection of inputs that can be placed in the current set. The Untrimmed Input Set contains the set of inputs that do not appear in any earlier set. In most cases the Untrimmed Input Set, and the Available Inputs list will be the same. These two lists differ when the partition contains a sequence of identical numbers. The Current Input Set is the list of inputs that have actually been selected for the current set.

Allocate an integer array \(P\) of size \(n\)
Set \(P[0]\) equal to \(n\)
For \(i=1\) through \(n\), Set \(P[i]\) equal to \(0\)
Print \(P\);
While \(P[n-1]\) Not Equal to \(1\)
  Search \(P\) backwards starting from \(P[n-1]\) to \(P[0]\)
  searching for an element greater than \(2\)
  Set \(Pos\) equal to the position of this element
  Decrement \(P[Pos]\) by \(1\)
  Set \(Total\) equal to the sum of \(P[0]\) through \(P[Pos]\)
  Set \(Residual\) equal to \(n\) minus \(Total\)
  Set \(Limit\) equal to \(P[Pos]\)
  For \(i=Pos+1\) through \(n-1\)
    Set \(P[i]\) equal to the min of \(Limit\) and \(Residual\)
  Subtract \(P[i]\) from \(Residual\)
EndWhile
Print \(P\);
EndFor
Figure 16. Generating the Partitions of a Number.

For \(i=0\) to \(n-1\)
  \(Array[i] = i\)
EndFor
Done = False;
While Not Done
  Print \(Array\)
  \(i = m-1\)
  \(Limit = n-1\)
  While \(i>=0\) And \(Array[i]<Limit\)
    \(Limit = Limit-1\)
    \(i = i+1\)
  EndWhile
  If \(i>=0\) Then
    \(Array[i] = Array[i] + 1\)
    \(i = i+1\)
  EndIf
Else
  Done = True
EndIf
EndWhile
Figure 17. Generating Successive Combinations.

When a partition has several numbers greater than \(1\), the procedure is more complex. Consider, for example, the partition of 10, 4,3,2,1. To generate the set of four inputs, we must compute the combinations of ten things taken four at a time. For each one of these combinations, we must select sets of three from the remaining inputs by generating all combinations of six things taken three at a time. Then for each pair of sets containing four and three inputs, we must generate a set of two from the remaining inputs, generating all combinations of three things taken two at a time. The total number of partial symmetries defined by this partition is given by the following formula.

\[
\begin{pmatrix} 10 \\ 4 \\ 6 \\ 3 \\ 2 \end{pmatrix} = 210 \times 20 \times 3 = 12,600
\]

When the partition has several equal digits greater than two, the situation is even more complicated. Consider the partition of 5: 2,2,1.
For example, consider the partition of 10: 4,3,2,1. Three data structures will be allocated for this partition, and they will be initialized as shown in Figure 20. Since this partition does not contain consecutive identical numbers, the Available Inputs and the Untrimmed Input Set are the same for all three data structures.

When generating consecutive partial symmetries, the last data structure in the list is advanced until there are no more available combinations. The Current Input Set is advanced using the algorithm of Figure 17. For the data structures of Figure 20, the Current Input Set of the final data structure will be advanced from 8,9 to 8,10 and then to 9,10. Once this combination has been generated, the algorithm backtracks to the preceding data structure and advances it using the algorithm of Figure 17. (The advancement is always done with respect to the Available Inputs list. The Untrimmed Input Set does not participate in this operation.)

When backtracking into the second data structure of Figure 20, the algorithm will advance the Current Input Set to 5,6,8, and will then reinitialize the third data structure. The new Available Input list will be 7,9,10, and the Untrimmed Input Set will be identical. The Current Input Set will be initialized to 7,9, and the selection will proceed as before. Eventually the algorithm will backtrack into the first data structure, and once this data structure has been advanced to 7,8,9,10, the process will terminate with the next backtrack into the first data structure.

When there are identical numbers in the input list, the advancement and backtracking operations proceed exactly as before, but the initialization of the data structures is different. The difference lies in the initialization of the Available Inputs list. Consider the partition of 11: 3,3,2,2,1. This partition will cause four data structures to be created as shown in Figure 21. This array of data structures is handled in a way that is almost identical to that of Figure 20, but during initialization, additional operations will be performed for the second and fourth data structures. These data structures correspond to the second 3 and the second 2 in the partition.

When initializing any data structure, the first step is to create the Untrimmed Input Set. For the first data structure in the array, the Untrimmed Input Set is initialized to a complete set of inputs for the function. The Untrimmed Input Set is copied to the Available Inputs list and no further operations are performed.

For subsequent data structures, the Untrimmed Input Set is copied from the previous data structure, and all inputs that appear in the Current Input Set of the preceding data structure are deleted. The Untrimmed Input Set is then copied to the Available Inputs list. Next it is necessary to examine the corresponding partition numbers of the current data structure and its predecessor. If they are different, no further operations are performed. If they are the same then the algorithm discards the Available Inputs list of the current data structure, and replaces it with the Available Inputs list of the preceding data structure. As before, any inputs of the Current Input Set of the preceding data structure are deleted. In addition, the algorithm obtains, $s$, the smallest input from the Current Input Set of the preceding data structure. The Available Inputs list of the current data structure is then trimmed again by deleting any elements smaller than $s$. The Untrimmed Input Set of the current data structure is unaffected by these operations.

Once the Available Inputs list of a data structure has been computed, the first $k$ inputs of the Available Inputs list are copied to the Current Input Set, where $k$ is the partition number corresponding to the current data structure. When there are identical consecutive numbers in the partition, it is possible that the Available Inputs list will have fewer than $k$ inputs. If this occurs then the initialization process is terminated and the backtracking process is initiated.

The data structures of Figure 21 show the effects of the additional trimming operations. It is assumed that the data structures have been advanced several times to get to this state. Note that inputs 1 and 2 have been trimmed from the Available Inputs list of the second data structure, and that 1 has been trimmed from the Available Inputs list of the fourth data structure. These extra trimming operations guarantee that the algorithm will generate the partial symmetry $(1,2,3)(4,5,6)(7,8)(9,10)$, but never $(4,5,6)(1,2,3)(7,8)(9,10)$ or $(1,2,3)(4,5,6)(9,10)(7,8)$. The pseudo-code for the initialization process is given in Figure 22, while the advancement algorithm is given in Figure 23.
Read partition \(P\);
\(k = \text{Count}\) elements greater than 1 in \(P\);
Allocate Array of Enumeration Structures, \(A[k]\);
\(A[0].\text{Available} = \{1, \ldots, n\}\);\n\(A[0].\text{Untrimmed} = \{1, \ldots, n\}\);
Copy first \(P[0]\) elements from \(A[0].\text{Available}\) to \(A[0].\text{Current}\);
\text{For} \(i = 1\) through \(n-1\) \text{ Do}
  \begin{align*}
  &\text{Copy} \ A[i-1].\text{Untrimmed} \text{ to} \ A[i].\text{Untrimmed}; \\
  &\text{Delete} \text{ all elements in} \ A[i-1].\text{Current} \text{ from} \ A[i].\text{Untrimmed}; \\
  &\text{If} \ P[i-1] = P[i] \text{ Then} \\
  &\quad \text{Copy} \ A[i].\text{Untrimmed} \text{ to} \ A[i].\text{Available}; \\
  &\text{Else} \\
  &\quad \text{Copy} \ A[i-1].\text{Available} \text{ to} \ A[i].\text{Available}; \\
  &\quad \text{Delete} \text{ all elements in} \ A[i-1].\text{Current} \text{ from} \ A[i].\text{Available}; \\
  &\quad S = \text{smallest element in} \ A[i-1].\text{Current}; \\
  &\quad \text{Delete} \text{ all elements smaller than} \ S \text{ from} \ A[i].\text{Current}; \\
  \end{align*}
\text{EndIf}
\text{EndFor}
\text{Figure 22. The Partial Symmetry Initialization Algorithm.}

\begin{align*}
&i = n-1; \\
&\text{While} \ \text{AtLimit}(A[i].\text{Available}) \ \text{And} \ \ i > 0 \ \text{Do} \\
&\quad i = i-1; \\
&\text{EndWhile} \\
&\text{If} \ i < 0 \ \text{Then} \\
&\quad \text{Exit}; \\
&\text{EndIf} \\
&\text{Advance} \ A[i].\text{Available}; \\
&\text{For} \ j = i+1 \ \text{through} \ n-1 \ \text{Do} \\
&\quad \text{Initialize} \ A[j]; \\
&\text{EndFor} \\
&\text{Figure 23. The Partial Symmetry Advancement Algorithm.}
\end{align*}

After each advancement, the “Current Input Set” element of each data structure has a set of inputs. These sets of inputs define one partial symmetry. A symmetry rule is created, and a set of orbits is generated from the array of data structures. This is done by starting with an orbit set that contains one orbit for each vector. For each data structure, the “Current Input Set” is used with each existing pair of orbits. If there is a pair of vectors, one from each orbit, such that the bits in the positions specified by the “Current Input Set” can be rearranged to create the other vector, then the two orbits are combined into a single orbit. Once all data structures in the array have been processed, the remaining orbits are the Boolean orbits of the partial symmetry.

The pseudo-code for generating the Boolean orbits from partial symmetries is given in Figure 24. To test whether rearranging specific positions can transform one vector into another, we zero the specified positions in each vector. The two resulting vectors must be equal. Next we zero everything except the specified positions. The resulting vectors must have the same number of 1’s. (i.e. be of the same weight.)

\text{SymmetryRule.\text{ OrbitSet} = One Orbit Per} \ n-\text{Element Vector; For} \ i=0 \ \text{through} \ n-1 \\
\text{For each pair of orbits} \ (O_j, O_k) \ \text{in} \ \text{SymmetryRule.\text{ OrbitSet}} \\
\begin{align*}
&\text{For each vector pair} \ v \in O_j \ \text{and} \ w \in O_k \\
&\quad v' = \text{zero the} \ A[i].\text{Current} \text{ Positions of} \ v; \\
&\quad w' = \text{zero the} \ A[i].\text{Current} \text{ Positions of} \ w; \\
&\quad v'' = \text{zero all except} \ A[i].\text{Current} \text{ positions of} \ v; \\
&\quad w'' = \text{zero all except} \ A[i].\text{Current} \text{ positions of} \ w; \\
&\text{If} \ v' = w' \ \text{and} \ \text{weight}(v') = \text{weight}(w'') \ \text{Then} \\
&\quad \text{Add} \text{ all Vectors of} \ O_k \ \text{to} \ O_j; \\
&\text{Delete} \ O_k \ \text{from} \ \text{SymmetryRule.\text{ OrbitSet};} \\
&\text{Exit For; // innermost “For”} \\
&\text{EndIf} \\
&\text{EndFor} \\
&\text{EndFor} \\
&\text{EndFor} \\
&\text{Figure 24. Generating Boolean Orbits for a Partial Symmetry.}
\end{align*}

9 Extended Symmetric Variable Pairs
Researchers have identified various types of symmetric variable pairs that go beyond those discussed in Section 2. These extended types are defined in terms of the cofactors of a function \(f\), with respect to a pair of variables \(x_i, x_j\). As before, these are designated \(f_{00}, f_{01}, f_{10}\), and \(f_{11}\). The extended types of symmetry are defined by relations between these four cofactors. Figure 25 lists the six possible relations, and the types of symmetry defined by each.

\begin{tabular}{|c|c|}
\hline
Relation & Type \\
\hline
\(f_{01} = f_{10}\) & Ordinary \\
\(f_{00} = f_{11}\) & Multi-Phase \\
\(f_{00} = f_{01}\) & Single Variable \\
\(f_{00} = f_{10}\) & Single Variable \\
\(f_{01} = f_{11}\) & Single Variable \\
\hline
\end{tabular}

Figure 25. The Extended Symmetry Relations.

It is possible to create Boolean orbits for each of these six relations. Let us assume that we wish to detect these symmetries in a four-input function \(f\), with respect to the variables \(x_i\) and \(x_j\). The Boolean orbits for these six relations are given in Figure 26.

\begin{tabular}{|c|c|c|}
\hline
Relation & \(f_{01} = f_{10}\) & \(f_{00} = f_{11}\) & \(f_{00} = f_{01}\) \\
\hline
\(f_{01} = f_{10}\) & 0100 1000 & 0000 1100 & 0000 1000 \\
\(f_{00} = f_{11}\) & 0101 1001 & 0001 1101 & 0001 1001 \\
\(f_{00} = f_{01}\) & 0110 1010 & 0010 1110 & 0010 1010 \\
\(f_{11} = f_{10}\) & 0111 1011 & 0111 1111 & 0111 1011 \\
\hline
\(f_{00} = f_{10}\) & 0000 1000 & 0010 1100 & 1000 1100 \\
\(f_{00} = f_{11}\) & 0001 1001 & 0101 1101 & 1001 1101 \\
\(f_{01} = f_{11}\) & 0010 1010 & 0110 1110 & 1010 1110 \\
\hline
\end{tabular}

Figure 26. Boolean Orbits for Extended Relations.

As pointed out in [Maurer. 2011], the single-variable relations are a special case of conjugate symmetry, and are best handled through matrix-based symmetry. Multi-phase symmetry is discussed more thoroughly in the following section.
10 Multi-Phase Symmetry

Multi-phase symmetry is the same as ordinary symmetry with one or more inputs inverted with respect to the others. In other words, the inputs of the function are assumed to be a mixture of active-high and active-low inputs. Multi-phase symmetry is normally defined in terms of pairs of symmetric variables, but like ordinary symmetry, it can be defined entirely by Boolean orbits. Suppose we wish to detect a multi-phase symmetry between the first two variables of a four-input function, \( f \). In terms of cofactors, we need to verify that \( f_{00} = f_{11} \). This is just the Shannon relation, \( f_{10} = f_{01} \), with one of the inputs inverted (it doesn’t matter which one). The Boolean orbits of this relation are given in Figure 27, along with the Boolean orbits of an ordinary symmetry in the same two variables. (These are the same orbits given in Figure 26.)

| Multi-Phase | Ordinary |
|-------------|----------|
| 0000 1100   | 0100 1000 |
| 0001 1101   | 0101 1001 |
| 0010 1110   | 0110 1010 |
| 0011 1111   | 0111 1011 |

Figure 27. The Boolean Orbits of a Multi-Phase Symmetry.

Figure 27 shows that we can obtain the multi-phase Boolean orbits by inverting one of the inputs of the ordinary Boolean orbits. In fact, we can invert any subset of the inputs, but not all such inversions will change the orbits. For example, inverting the last two bits of the orbits of Figure 27 will leave them unchanged.

In terms of vectors, the multi-phase orbits of Figure 27 were obtained by adding a constant vector, 0100, to each vector of each orbit. We define adding a vector to an orbit as follows. If \( S \) is an orbit and \( v \) is a vector, then \( S + v = \{w + v | w \in S\} \). We can apply this principle to any set of orbits, not just those representing symmetric variable pairs. Consider the symmetry group \( \{I,(1,2)(3,4)\} \). This type of symmetry cannot be defined in terms of symmetric variable pairs. (The function \( x_1x'_2 + x_2x'_1 \) from Section 2 possesses such symmetry.)

We can also detect multi-phase symmetry in an arbitrary number of variables, by adding an arbitrary vector to each of the orbits of the original symmetry. Let us assume that we wish to detect symmetry of type \( I,(1,2)(3,4) \) in a function whose first and last variables are active high. To do this, we must add the vector 1001 to all orbits of the original symmetry. Figure 28 shows how this is done.

| Original     | Multi-Phase “1001” |
|--------------|---------------------|
| 0001 0010    | 1000 1011 |
| 0100 1000    | 1101 0001 |
| 0101 1010    | 1100 0011 |
| 0110 1001    | 1111 0000 |
| 0111 1011    | 1110 0010 |
| 1101 1110    | 0100 0111 |

Figure 28. Complex Multi-Phase Symmetry.

Computation of multi-phase orbits can be performed on the fly or loaded from a library. We normally compute the multi-phase orbits on the fly from existing libraries.

As a consequence of Theorem 7, there are always (at least) two vectors that will produce the same set of orbits. If \( v + w = 111...1 \), the vector of all 1’s, then adding \( v \) to a set of orbits will produce the same result as adding \( w \) to the same orbits. In practice, we take advantage of this by using only those vectors whose first element is zero.

11 Anti-Symmetry

Anti-symmetry [Tsai and Marek-Sadowska. 1994] (which is also known as Skew Symmetry and Negative Symmetry) was initially defined in terms of symmetric variable pairs. As with other symmetries, the anti-symmetries are defined with respect to the four cofactors \( f_{00}, f_{01}, f_{10}, \) and \( f_{11} \), taken with respect to the variables \( x_i \) and \( x_j \). The normal symmetric variable pairs are defined by the relations in Figure 25. The corresponding anti-symmetry relations are given in Figure 29.

| Relation   | Type          |
|------------|---------------|
| \( f_{01} = f_{10} \) | Ordinary      |
| \( f_{00} = f_{11} \) | Multi-Phase   |
| \( f_{00} = f_{01} \) | Single Variable |
| \( f_{01} = f_{00} \) | Single Variable |
| \( f_{01} = f_{10} \) | Single Variable |
| \( f_{01} = f_{11} \) | Single Variable |

Figure 29. The Anti-Symmetry Relations.

Anti-Symmetry is important, because it is just as common as normal symmetry. For example, we analyzed all 4-input functions and determined that in this set of functions there are 24,576 examples of ordinary symmetric variable pairs, and the same number of ordinary anti-symmetric variable pairs.

Suppose \( f \) has an anti-symmetric variable pair \((x_i, x_j)\). If this were an ordinary symmetric variable pair, the symmetry group of \( f \) would contain the group \( \{I,(x_i, x_j)\} \). To test for this subgroup we would check the orbits \{01x...x,0x...x\}, where “x...x” ranges through all bit combinations. However, because this is an anti-symmetric variable pair, \( f_{00} \) and \( f_{11} \) must produce the opposite value for each input. If \( 01x...x \) produces a one value then \( 0x...x \) must produce a zero value, and vice-versa. Thus, no orbit may imply either \( f \) or \( f' \). This is a necessary and sufficient condition for the anti-symmetric variable pair to exist. Thus, we can test for anti-symmetric variable pairs using the same library used to detect normal symmetric variable pairs. When we test the orbits, we must test for non-implication rather than implication. This is a simple change, enabling the algorithm to detect anti-symmetric pairs just as easily as normal symmetric pairs.

This principle extends to anti-symmetric pairs of all six types.

12 Kronecker Symmetry

As with multi-phase and anti-symmetries, the Kronecker symmetries [Chrzanoska-Jeske. 1999] of a Boolean function \( f \) are defined with respect to two input variables \( x_i \) and \( x_j \). Let \( f_{00} \), \( f_{01} \), \( f_{10} \), and \( f_{11} \) be the cofactors of \( f \) with respect to these two variables. There are five types of Kronecker symmetry, defined by the five relations given in Figure 30. In this figure, \( \hat{0} \) represents the constant zero function.

Suppose we are dealing with a 4-input function with input variables \( x_1 \), \( x_2 \), \( x_3 \), and \( x_4 \). The Boolean orbits of the Kronecker symmetries are given in Figure 31.
\[
\begin{align*}
    f_{a_0} \oplus f_{a_0} \oplus f_{i_1} &= \hat{0} \\
f_{a_0} \oplus f_{a_0} \oplus f_{i_1} &= \hat{0} \\
f_{a_0} \oplus f_{i_0} \oplus f_{i_1} &= \hat{0} \\
f_{a_0} \oplus f_{i_0} \oplus f_{i_0} &= \hat{0} \\
f_{a_0} \oplus f_{i_0} \oplus f_{i_0} \oplus f_{i_1} &= \hat{0}
\end{align*}
\]

Figure 30. The Kronecker Symmetries.

\[
\begin{align*}
    f_{a_0} \oplus f_{a_0} \oplus f_{i_1} &= \hat{0} \\
f_{a_0} \oplus f_{a_0} \oplus f_{i_0} \oplus f_{i_1} &= \hat{0} \\
f_{a_0} \oplus f_{i_0} \oplus f_{i_0} \oplus f_{i_1} &= \hat{0} \\
f_{a_0} \oplus f_{i_0} \oplus f_{i_0} \oplus f_{i_0} &= \hat{0} \\
f_{a_0} \oplus f_{i_0} \oplus f_{i_0} \oplus f_{i_0} \oplus f_{i_1} &= \hat{0}
\end{align*}
\]

Figure 31. Boolean Orbits for Kronecker Symmetries.

As with other Boolean orbits, we reduce each orbit to a characteristic function. Because the vectors of each orbit are not required to be mapped to the same value, we cannot use the implication relation to detect Kronecker symmetries. Instead we compute the function \( D = f \& C_i \) for each orbit \( O_i \), and count the number of input vectors that are mapped to 1 by \( D_i \). (The result cannot exceed the number of vectors in the orbit.) If the result is an even number for each orbit, then \( f \) possesses the associated Kronecker symmetry. Note that the function \( D_i \) can be used to compute the implication relation, since \( C_i \) implies \( f \) if and only if \( D_i = C_i \), and implies \( \overline{f} \) if and only if \( D_i = \hat{0} \).

There are also symmetries known as the Negative Kronecker symmetries, defined by the relations of Figure 32, where \( \overline{1} \) represents the constant-one function.

\[
\begin{align*}
    f_{a_0} \oplus f_{a_0} \oplus f_{i_1} &= \overline{1} \\
f_{a_0} \oplus f_{a_0} \oplus f_{i_0} &= \overline{1} \\
f_{a_0} \oplus f_{i_0} \oplus f_{i_0} &= \overline{1} \\
f_{a_0} \oplus f_{i_0} \oplus f_{i_0} &= \overline{1} \\
f_{a_0} \oplus f_{i_0} \oplus f_{i_0} \oplus f_{i_1} &= \overline{1}
\end{align*}
\]

Figure 32. The Negative Kronecker Symmetries.

We proceed exactly as before, but in this case, once we have obtained \( D_i \), and counted the vectors that map to 1, we check for an odd number instead of an even number.

13 Matrix-Based Symmetry

As pointed out in [Maurer. 2011], it is possible to characterize the symmetry of an \( n \)-input Boolean function in terms of \( n \times n \) permutation matrices instead of degree-\( n \) permutations. When this is done, the matrices are assumed to be over the field \( GF(2) \), which has two elements, zero and one, and which uses the AND and XOR functions in place of multiplication and addition. The advantage of doing this is that any group of \( n \times n \) non-singular matrices can be used as a symmetry group, and there are many more matrix groups than there are permutation groups. The number of degree-\( n \) permutations is \( n! \), while the number of \( n \times n \) non-singular matrices is given by the following formula.

\[
G(n) = \prod_{i=0}^{n-1} (2^n - 2^i)
\]

Figure 33 shows a comparison of the size of these numbers for values of \( n \) from 2 through 8.

| \( n \) | \( n! \) | \( G(n) \) |
|---|---|---|
| 2 | 2 | 6 |
| 3 | 6 | 168 |
| 4 | 24 | 20,160 |
| 5 | 120 | 9,999,360 |
| 6 | 720 | 20,158,709,760 |
| 7 | 5,040 | 163,849,992,929,280 |
| 8 | 40,320 | 5,348,063,769,211,699,200 |

Figure 33. Number of Permutations vs. Matrices.

Because of the large number of non-singular matrices, the opportunities to detect symmetry are much greater. There are many types of matrix-based symmetry that have no counterparts in permutation-based symmetry. Conjugate symmetry [Maurer. 2011] is just one of these.

Just like permutation-based symmetry, matrix-based symmetry can be characterized in terms of Boolean orbits. The Boolean orbits of a matrix group can be computed using the same algorithm as for permutation groups. Figure 34 shows a matrix group that has no counterpart in permutations, and gives the Boolean orbits for it.

\[
\begin{pmatrix}
0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\
1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\
1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1
\end{pmatrix}
\]

000
001 111 110
010 100 101
011

Figure 34. A Matrix Group and Its Boolean Orbits.

Like permutation groups, the symmetry generated by a matrix group is completely determined by its Boolean orbits. Theorem 5 applies to matrix groups as well as to permutation groups, so the same principle can be used to detect symmetry with respect to matrix groups. Of course, the set of all matrix groups is much too large to permit a library of all possible subgroups to be constructed. It is therefore necessary to focus on known symmetry types such as conjugate symmetry and the subgroups of these symmetries. For such symmetry types, it is possible to generate the library entries on the fly using the standard libraries as a basis.

14 Other Types of Symmetry

We have designed dynamic generators for rotational and dihedral symmetry. The symmetry groups in question are \( R_n \) for rotational symmetry of degree \( n \), and \( D_n \) for dihedral symmetry of degree \( n \). The output from these generators is normally used on the fly and discarded, but it could easily be placed in a library for future use. To create a library entry for rotational symmetry we start with an \( n \)-element vector and rotate it \( n-1 \) times to create each orbit.

Figure 35 shows the orbits for rotational symmetry of degree 5.
For \( n \leq 6 \) rotational and dihedral symmetry are the same. Dihedral symmetry includes all \( n \) rotations of a vector, but also includes the mirror image of a vector. (The mirror image of \( (v_1, v_2, v_3, v_4) \) is \( (v_4, v_3, v_2, v_1) \).) For \( n < 6 \), the mirror image of a vector can be obtained by rotating the vector. However, for \( n = 6 \) and larger, there are vectors whose mirror image cannot be obtained in this fashion. For 6 inputs, the vector 001011 has a mirror image of 110100, but no rotation of 001011 will produce 110100. This means that for \( n = 6 \) and larger, that dihedral symmetry and rotational symmetry are distinct.

To create an entry for dihedral symmetry of degree \( n \), we first start with rotating each vector \( n - 1 \) times, and then we reverse each vector to produce the required orbits.

Another type of symmetry, that is used to simplify Boolean functions, is Auto Symmetry. Auto symmetry occurs when an \( n \)-input Boolean function becomes an \( n-k \)-input function under a linear transformation of its inputs [Bernasconi, et al. 2008].

As explained in [Bernasconi, et al. 2008] autosymmetric functions take constant values on a subspace of \([0,1]^n\) and its affine spaces. The subspace and its affine spaces constitute the Boolean orbits of the autosymmetry. Autosymmetry is detected in the same way as ordinary symmetry. Figure 36 gives an example of a set of autosymmetry orbits. Autosymmetry orbits are always all the same size. Every element of \([0,1]^n\) appears in one of the orbits, and the size of the orbits is a power of 2.

\[
\begin{align*}
0000 & 0010 & 0100 & 1000 & 0100 & 0010 & 0001 & 0011 & 0101 & 0110 & 1001 & 1010 & 1100 & 0111 & 1011 & 1101 & 1110 & 0100 & 1000 & 0011 & 0111 & 1101 & 1111 & 0010 & 0110 & 1010 & 1110 & 0001 & 0011 & 0101 & 0111 \n\end{align*}
\]

Figure 36. Autosymmetry Boolean Orbits.

Although hierarchical symmetry has been extensively studied [Kravets and Sakallah, 2000, Kravets and Sakallah, 2002], we do not treat it as a distinct type of symmetry. As with other symmetries, hierarchical symmetries are specified as groups of permutations. Our existing libraries contain many examples of hierarchical symmetry groups.

### 15 Experimental Data

We ran many experiments to test the efficacy of the USD algorithm. Initially, we focused on analyzing all functions with 5 or fewer inputs. Functions with 2 and 3 inputs are trivial, because a 2-input function is either non-symmetric or totally symmetric. A 3-input function is non-symmetric, totally-symmetric, or partially symmetric in two variables. The total number of each such function is given in Figures 37 and 38.

| Type            | Count |
|-----------------|-------|
| Totally Symmetric | 8     |
| Non-Symmetric   | 8     |

Figure 37. All 2-input Functions.

Four-input functions are more interesting. In addition to total and partial symmetries, there is dihedral symmetry and lock-step symmetry between two variable pairs. We use the term lock-step symmetry to describe non-independent sub-symmetries between two sets of variables. The only example of this for 4-input functions is exemplified by the function \( x_1'x_2 + x_3'x_4 \) from Section 2. The symmetry group for this function is \{1,(1,3)(2,4)\}, where the two transpositions \( (1,3) \) and \( (2,4) \) must operate in lock-step with one another. For more than 4 inputs, this type of symmetry is much more varied. The results of our analysis of 4-input functions are given in Figure 39. In this figure, we combine results for the symmetries that are conjugate to one another. The number of conjugates is given in parentheses following the symmetry type. Each function is counted only once. There are subgroups of \( S_4 \), such as \( K_4 \) and \( A_4 \), which are not listed, because they are forbidden.

| Type                  | Count  |
|-----------------------|--------|
| Total Symmetry        | 32     |
| \( D_4 \) dihedral (3) | 96     |
| Three-variable Partial (4) | 896 |
| (2,2) Partial (3)     | 1,344  |
| (2,2) Lock-Step (3)   | 1,344  |
| Two-variable Partial (6) | 18,816 |
| Non-Symmetric         | 43,008 |

Figure 39. All 4-input Functions

As Figure 39 shows, the proportion of symmetric functions to non-symmetric functions decreases as the number of inputs increases. The results of our analysis for all 5-input functions is given in Figure 40.

| Type                  | Count|
|-----------------------|------|
| Total Symmetry        | 64   |
| Four Variable Partial (5) | 4,800 |
| (3,2) Partial (10)   | 40,320 |
| \( D_5 \) Dihedral (6) | 1,152 |
| \( D_4 \) Dihedral sub-symmetry (15) | 46,080 |
| (2,2) Partial (15)   | 3,749,760 |
| \( K_4 \) sub-symmetry (5) | 30,720 |
| (2,2) Lock-Step (15) | 11,606,400 |
| Three Variable Partial (10) | 595,200 |
| Two-Variable Partial (10) | 158,204,160 |
| Non-Symmetric         | 4,120,688,640 |

Figure 40. All 5-input Functions

It is not possible to obtain a complete analysis of all \( n \)-input functions, where \( n \geq 6 \). Even for \( n = 6 \), such an analysis would require many of thousands of years on existing hardware. This is unfortunate, because the number of different types of symmetry explode for six or more inputs. For five and fewer inputs, the types of symmetry are more-or-less what one would expect. For six and more inputs, the
variety is astounding, and some types are less than intuitive and quite difficult to describe. For example, $S_4$ contains a class of subgroups which is isomorphic to the wreath product of $S_2$ and $S_1$, a class that is the split extension of $S_1 \times S_1$ by $S_1$, and another that is the non-split extension of $S_1$ by $S_1$. In $S_4$ we have a class that is isomorphic to the quaternion group. (These are only a few examples out of many.) One needs some expertise in group theory even to understand what these groups are. Describing their effects is quite difficult.

Figure 40 reinforces the idea that as the number of variables increases, the proportional number of non-symmetric functions decreases. However, this does not necessarily describe what happens with functions that are used in practice. To determine what sort of functions one would encounter in practice, we conducted experiments with the ISCAS85, LGSynth89 and LGSynth91. The initial results for ISCAS85 were uninformative because the standard description of these circuits is in terms of individual gates whose symmetries are obvious. To obtain more substantive results we partitioned the ISCAS85 circuits into larger groups and treated each group as a single function. The method for doing this was to identify the fanout-free regions of each circuit, and treat each such region as a function. (A fanout-free circuit has a single output, and can easily be treated as a single function.) Because we do not have complete libraries for circuits with more than 8 inputs, we further partitioned the fanout-free regions into subcircuits containing no more than 8 inputs. We expected this procedure to produce mostly partial and total symmetries, and this was indeed the case, but there were also some interesting surprises. Figure 41 summarizes the results for total and partial symmetry and gives the counts of the interesting cases.

| Type                          | Count |
|-------------------------------|-------|
| Total Symmetry                | 2,674 |
| Single Partial Symmetry       | 923   |
| Multi-Partial Symmetry        | 193   |
| Non Symmetric                 | 191   |
| $D_4$ Dihedral                | 146   |
| $D_4$ Dihedral $\times S_2$  | 12    |
| $D_4$ Dihedral $\times (2,2)$ lock step | 4 |
| K4 sub-symmetry               | 3     |
| (2,2) Lock-Step               | 2     |
| $S_4 \times (2,2)$ Lock-Step  | 1     |

Figure 41. The ISCAS85 Benchmarks

To further study the symmetries of (more or less) real circuits, we ran tests on the LGSynth89 and LGSynth91 benchmarks. The results of these tests are given in Figure 42. Figures 41 and 42 clearly show that the circuits used in practice contain far more symmetries than randomly chosen circuits. This emphasizes the importance of correctly detecting the symmetries of a function.

The USD is implemented as a collection of objects representing, functions, orbits, symmetry rules (a group and its orbits), rule libraries and function libraries. The low-level implementations were hidden from the user, to enable many different implementations to be used without affecting the high-level algorithms. The only requirement for the USD algorithm to function correctly is the ability to compute the implication relation. For compressed libraries and sub-symmetries it is also necessary to compute cofactors and the product of a permutation with a function.

In the current implementation we model functions as compressed truth tables, which is an array of 64-bit integers with one bit for each input vector. This implementation is extremely efficient for up to six inputs, but rapidly becomes less efficient as the number of inputs increases beyond this point. A single 64-bit integer will suffice for up to six inputs. Beyond six inputs, the number of integers doubles for each input, with 16-20 inputs being the practical limitation. For more general circuits, Binary Decision Diagrams or something similar would almost certainly more efficient.

We ran all experiments on modest hardware: a Dell laptop containing an Intel P9500 Core 2 Duo 2.53Ghz CPU with 3.48 Gigabytes of RAM and Windows XP Professional with Service Pack 3. To gauge the efficiency of the algorithm, we measured the amount of real time required to determine the symmetry of 1,000,000 functions. The results are reported in Figure 43. For three and four input functions, it was necessary to test the same functions repeatedly. For five, six and seven inputs, the first 1,000,000 functions were tested. It should be noted that the efficiency of the algorithm depends heavily on the underlying implementation, so these numbers should be taken only as rough guidelines.

The superior speed of the USD algorithm is obvious for 3-6 inputs. The superior speed is also evident for the 7-input test when one remembers that the conventional algorithm is testing for 877 different symmetries while the USD algorithm is testing for 11,300 different symmetries. (Twelve times as much work for three and a half times as much time.)

| Number of Inputs | USD       | Conventional |
|------------------|-----------|--------------|
| 3                | 0.043079  | 0.128639     |
| 4                | 12.464    | 258.660      |
| 5                | 30.420    | 433.378      |
| 6                | 246.751   | 640.029      |
| 7                | 3189.840  | 893.361      |

Figure 42. The LGSynth89, and LGSynth91 Benchmarks

Figure 43. Seconds per 1,000,000 Functions.

The categorization of all 4-input functions was virtually instantaneous, as were the experiments with the ISCAS85, LGSynth89 and LGSynth91 benchmarks. The categorization of all 5-input functions took about four hours.

For eight or more inputs, the conventional approach will normally be faster than the USD algorithm. However the results will be considerably less satisfactory, since only a small fraction of the available symmetries will be detected.

We again caution the reader that these results are valid only for our current implementation of the USD algorithm. It is extremely likely that much more efficient implementations will be developed in the future, necessitating a new characterization of the algorithm’s performance.
16 Conclusion

The USD algorithm is a powerful tool that can be used in many different contexts. It is a simple, yet powerful and efficient algorithm for detecting virtually any type of symmetry. It is our belief that many types of symmetry could be exploited if there were methods to detect them. Because the USD algorithm makes these types accessible, we expect to see significantly more exploitation of symmetry in the future.

The initial paper on detecting symmetry in Boolean functions was published in 1949 [Shannon. 1949], and since that time the basic principle introduced in this paper has been refined and improved many times. We believe that the USD algorithm is similar to [Shannon. 1949] in that it is a beginning rather than an end. We hope that the principles elucidated here will be refined and improved in the same way that Shannon’s principle has been refined and improved over the years.

One area where we hope to see improvement is in the parallelization of the USD algorithm. The USD algorithm does many comparisons, virtually all of which are independent of one another. Library searches could easily be parallelized, because searching one entry is only loosely connected to searching other entries. While it is true that our algorithm searches the largest subgroups first, these searches tend to be faster than the searches for smaller subgroups because there are fewer Boolean orbits. This would allow the faster parallel searches to abort the slower ones once a match is found. Searching an individual library entry could also be easily parallelized since the testing of one Boolean orbit does not depend on the outcome of any other test. When large numbers of functions are being tested, each function can be tested in parallel with the others. We believe that the USD algorithm could be parallelized in such a way as to take advantage of virtually any available parallelism.

As for our own plans, the future development of the USD algorithm will include the incorporation of at least a portion of the $S_o$ through $S_n$ material, as well as the identification and incorporation of new parameterized symmetry types. In particular, we are interested in creating automatic generators for various types of matrix-based symmetry.

In any case, the USD algorithm should prove to be a powerful tool that can be used in many different areas of Electrical Design Automation.

SHANNON, C.E. 1949. The synthesis of two-terminal switching circuits. Bell System Technical Journal 28, 59-98.

ABDOLLAHI, A. 2006. Canonical form based boolean matching and symmetry detection in logic synthesis and verification.

BISWAS, N.N. 1970. On Identification of Totally Symmetric Boolean Functions. Computers, IEEE Transactions on 19, 645-648.

BORN, R.C. AND SCIDMORE, A.K. 1968. Transformation of switching functions to completely symmetric switching functions. IEEE Transactions on Computers 17, 596-599.

BUTLER, J.T., DUECK, G.W., SHMERKO, V.P. AND YANUSKEVICH, S. 2000. Comments on “Sympathy: fast exact minimization of fixed polarity Reed-Muller expansion for symmetric functions”. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 19, 1386-1388.

CHRZANOWSKA-JESKE, M. 2001. Generalized symmetric variables. In The 8th IEEE International Conference on Electronics, Circuits and Systems, Anonymous IEEE, 1147-1150.

CHUNG, K.S. AND LIU, C.L. 1998. Local transformation techniques for multi-level logic circuits utilizing circuit symmetries for power reduction. In Proceedings of the 1998 international symposium on Low power electronics and design, Anonymous ACM, 215-220.

DARGA, P.T., SAKALLAH, K.A. AND MARKOV, I.L. 2008. Faster symmetry discovery using sparsity of symmetries. In Proceedings of the 45th annual Design Automation Conference, Anonymous ACM, 149-154.

DRECHSLER, R. AND BECKER, B. 1995. Sympathy: fast exact minimization of fixed polarity Reed-Muller expressions for symmetric functions. In European Design and Test Conference, Anonymous IEEE, 91-97.

HU, B. AND MAREK-SADOSKA, M. 2001. In-place delay constrained power optimization using functional symmetries. In Design Automation and Test in Europe, Anonymous, 377-382.

HU, Y., SHIH, V., MAJUMDAR, R. AND HE, L. 2008. Exploiting Symmetries to Speed Up SAT-Based Boolean Matching for Logic Synthesis of FPGAs. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 27, 1751-1760.

KE, W. AND MENON, P.R. 1995. Delay-testable implementations of symmetric functions. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 14, 772-775.

KETTLE, N. AND KING, A. 2008. An anytime algorithm for generalized symmetry detection in ROBDDs. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 27, 764-777.

KRAVETS, V.N. AND SAKALLAH, K.A. 2002. Generalized symmetries in Boolean functions.

MAURER, P.M. 2011. Conjugate Symmetry. Formal Methods in System Design 38, 263-288.

MOHINKE, J., MOLITOR, P. AND MALIK, S. 2002. Limits of using signatures for permutation independent Boolean comparison. Formal Methods in System Design 21, 167-191.

MOLLER, D., MOHNKE, J. AND WEBER, M. 1993. Detection of symmetry of Boolean functions represented by ROBDDs. In IEEE International Conference on Computer-Aided Design, Anonymous IEEE, 680-684.

MUZIO, J.C., MILLER, D.M. AND HURST, S.L. 2008. Multivariable symmetries and their detection. IEE Proceedings on Computers and Digital Techniques 130, 141-148.
RICE, J. AND MUZIO, J. 2002. Antisymmetries in the realization of Boolean functions. In *IEEE International Symposium on Circuits and Systems*, Anonymous IEEE, , 69-72.

SCHOLL, C., MELCHIOR, S., HOTZ, G. AND MOLITOR, P. 1997. Minimizing ROBDD sizes of incompletely specified Boolean functions by exploiting strong symmetries. In *European Design and Test Conference*, Anonymous IEEE, , 229-234.

TSAI, C.C. AND MAREK-SADOWSKA, M. 1996. Generalized Reed-Muller forms as a tool to detect symmetries. *IEEE Transactions on Computers* 45, 33-40.

WANG, K.H. AND CHEN, J.H. 2004. Symmetry detection for incompletely specified functions. In *Proceedings of the 41st annual Design Automation Conference*, Anonymous ACM, , 434-437.

ZHANG, J.S., CHRZANOWSKA-JESKE, M., MISHCHENKO, A. AND BURCH, J.R. 2004. Generalized Symmetries in Boolean Functions: Fast Computation and Application to Boolean Matching. In *International Workshop on Logic Synthesis*, Anonymous, , 424-430.

TSAI, C.C. AND MAREK-SADOWSKA, M. 1994. Boolean matching using generalized Reed-Muller forms. In *Proceedings of the 31st annual Design Automation Conference*, Anonymous ACM, , 339-344.

KRAVETS, V.N. AND SAKALLAH, K.A. 2000. Generalized symmetries in Boolean functions. In *IEEE International Conference on Computer Aided Design*, Anonymous IEEE, , 526-532.

BRGLEZ, F., POWNALL, P. AND HUM, R. 1985. Accelerated ATPG and fault grading via testability analysis. In *Proceedings of IEEE Int. Symposium on Circuits and Systems*, Anonymous , 695-698.

MAURER, P.M. AND SCHAPIRA, A.D. 1988. A Logic-to-Logic Comparator for VLSI Layout Verification. *IEEE Transactions on Computer-Aided Design* 7, 897-907.

HOLT, D.F. 2010. Enumerating subgroups of the symmetric group. *Computational Group Theory and the Theory of Groups, II* 33-37.

PASSMAN, D.S. 1968. Permutation Groups. W. A. Benjamin, New York.

PFIEFFER, G. 2005. Subgroups of alternating and symmetric groups.

ROBINSON, D. 1995. A Course in the Theory of Groups. Springer, New York.

CHRZANOWSKA-JESKE, M. 1999. Generalized symmetric and generalized pseudo-symmetric functions. In *Proceedings of The 6th IEEE International Conference on Electronics, Circuits and Systems*, Anonymous IEEE, , 343-346.

BERNASCONI, A., CIRIANI, V., LUCCIO, F. AND PAGLI, L. 2008. Synthesis of Autosymmetric Functions in a New Three-Level Form. *Theory of Computing Systems* 42, 450-464.