Conjugate symmetry.

Article in Formal Methods in System Design · June 2011

DOI: 10.1007/s10703-011-0116-2 · Source: DBLP

Peter Maurer
Baylor University

Conjugate Symmetry

Peter M. Maurer

[1]***Abstract*****—Conjugate symmetry is an entirely new approach to symmetric Boolean functions that can be used to extend existing methods for handling symmetric functions to a much wider class of functions. These are functions that currently appear to have no symmetries of any kind. Such symmetries occur widely in practice. In fact, we show that even the simplest circuits exhibit conjugate symmetry. To demonstrate the effectiveness of conjugate symmetry we modify an existing simulation algorithm, the hyperlinear algorithm, to take advantage of conjugate symmetry. This algorithm can simulate symmetric functions faster than non-symmetric ones, but due to the rarity of symmetric functions, this optimization is only of limited benefit. Because the standard benchmark circuits contain many symmetries it is possible to simulate these circuits faster than is possible with the fastest known event-driven algorithm. The detection and exploitation of conjugate symmetries makes use of GF(2) matrices. It is likely that conjugate symmetry and GF(2) will find applications in many other areas of EDA.**

I. Introduction

This paper has two main aims. The first is the introduction of an entirely new approach to symmetric Boolean functions called *conjugate symmetry*. The second is to show how conjugate symmetry can be used to enhance the power of existing algorithms that take advantage of function symmetries. Our vehicle for doing this is a simulation algorithm called the hyperlinear algorithm[1] that uses function symmetries to accelerate the simulation of Boolean functions. Although conjugate symmetry has never been used anywhere in electronic design automation it has had some application in a number of other fields. (See, for example, [2] and [3].) Some of the material of this paper has appeared in abbreviated form in [4].

Symmetric functions have been studied for many decades by mathematicians (see [5], for example). However the systematic study of the mathematical properties of symmetric Boolean functions is relatively new in the field of electronic design automation. The earliest work that we know of is Shannon's paper on symmetries in relay switching circuits [6]. From this work we get the familiar concepts of total and partial symmetry, and a concept which we will call *Skew Symmetry* in this paper. This paper was the first design automation paper to use group theory to characterize Boolean function symmetries. A more comprehensive group-theoretic approach was used by the logic comparator LLC, which is described in [7]. This approach, the mathematical details of which can be found in [8], was the first to use hierarchical symmetries and generalized group symmetries. More recent discussions of hierarchical symmetry and generalized group symmetries can be found in references [9,10].

The general approach to symmetry is to divide symmetric Boolean functions into three primary categories, totally symmetric functions, partially symmetric functions and "other". The terminology for the third class varies depending on the publication, although in the mathematical world these types of functions are usually called *weakly symmetric*. (Some other terms are *G-Symmetry* [9], and *extended partial symmetry* [8]. There are several others.) For pragmatic reasons weak symmetries are often broken into two categories, hierarchical symmetries and general symmetries [8,9]. Recent work has identified other useful types of weak symmetry, such as rotational symmetry, dihedral symmetry, and set-symmetry. Since the number of types of symmetric functions grows with the number of inputs, there is an infinite number of different types of symmetry.

Reference [8] presents several useful points that are not addressed elsewhere. Orbits are computed with respect to input vectors rather than with respect to input variables. This categorization demonstrates that different symmetry groups can act identically on a Boolean function, making the two types of symmetry indistinguishable. Much recent work has focused on orbits with respect to the input variables of a function, but this approach has several drawbacks, the most obvious of which is the inability to distinguish between symmetries of the two functions $abcd$ and $ab+cd$, which are obviously different.

Symmetries can be categorized using the conjugacy relation, which distinguishes between symmetries that are truly different and those that are the same but applied to different inputs. For example the two functions $abc+c$ and $a+bcd$ have different symmetry groups, but the groups are conjugate to one another indicating that the two symmetries are the same, but applied to different inputs. On the other hand the symmetry groups of the two

[1] Manuscript received 11/??/09.
Peter M. Maurer is with the Department of Computer Science, Baylor University

functions $ab' + cd'$ and $ab + c + d'$ are not conjugate to one another even though they are isomorphic, indicating that the symmetries are fundamentally different. This categorization, along with the input-vector orbits, can be used enumerate all types of symmetry for a particular number of inputs. This categorization can be used to determine an accurate count of all functions possessing a certain type of symmetry.

The main point is that most Boolean functions possess *no symmetries of any kind*, no total symmetry, no partial symmetry, and no weak symmetry. Even though the number of symmetries increases with the number of inputs, the number of completely non-symmetric functions increases even faster. For the trivial case of three inputs, 98 of the 256 functions are completely non-symmetric. For four inputs 43,008 of the 65,536 functions are totally non-symmetric. As the number of inputs increases, symmetric functions *of any kind* become increasingly rare.

This brings us to the first objective of this paper. Up until this point, new types of symmetry have been defined by creating new subdivisions of the class of weak symmetries. Conjugate symmetry *does not do this*. Conjugate symmetry defines a super-class of symmetries, with *all known symmetries* as a relatively minor special case. In other words, conjugate symmetry does not concern itself with the 22,528 symmetric 4-input functions, but with the 43,008 non-symmetric 4-input functions. There are 32 totally symmetric 4-input functions, including the two constant functions. Adding skew symmetries increases the total to 228 totally symmetric functions while conjugate symmetry gives us 4576 totally symmetric functions. Combining skew and conjugate symmetry gives us 12,096 totally symmetric functions. As for partial symmetries, with respect conjugate symmetry alone there are 34,036 functions with at least a 3-input partial symmetry. In addition, all 65,536 functions have at least one pair of symmetric inputs.

Conjugate symmetry has the potential to extend the reach of existing symmetry detection algorithms such as those described in [12-14] to a vastly wider class of functions than was heretofore possible. Although conventional symmetries are rare, conjugate symmetries are not. As will be demonstrated in Section VII, conjugate symmetries are so common that it is virtually impossible to design a digital circuit without introducing at least some of them. To be fair, it has been pointed out that in *practical* circuits, conventional symmetry is more common than would be expected. However, conjugate symmetry can also be used to enhance the symmetry of functions that are already symmetric. Partially symmetric functions become totally symmetric, weakly symmetric functions become partially symmetric, and so forth.

To demonstrate the value of conjugate symmetry we have taken the HyperSim simulation algorithm[1], and enhanced it with the ability to detect conjugate symmetry. The changes are relatively small, but the theory behind why the changes work is fairly complex. We have chosen a simulation algorithm because simulation is the mainstay of most practical development projects. A substantial portion of the development time for a new product is consumed by simulation, and improvements in simulation speed can have a significant impact in terms of product quality and time-to-market. The performance of the HyperSim algorithm depends heavily on the ability to detect and use total and partial symmetries. The algorithm is independent of the representation of the functions and can work equally well with truth-tables, netlists, cube lists and BDD's. These properties make the HyperSim algorithm an ideal candidate for enhancement.

Of the many different simulation algorithms [15-33] that have appeared over the years, we have chosen the Inversion Algorithm for comparison with our enhanced version of HyperSim. The Inversion Algorithm is extremely fast, and competes well with Levelized Compiled Code simulation [18]. The speed of the Inversion Algorithm is due to its ability to exploit gate-level symmetries. This ability to handle low-level symmetries was the inspiration for developing HyperSim to exploit symmetries at the functional level.

The objective of the HyperSim algorithm was to provide a method for combining clusters of gates into a single simulation-unit that could be simulated as efficiently as a single gate. The main problem with the HyperSim approach is that gate simulations for non-symmetric functions can be extremely inefficient both in terms of time and space. For such functions, the number of operations required to perform a state transition depends on the number of inputs. Ordinary symmetries allow these operations to be reduced, but because symmetric functions are relatively rare, the opportunity to create efficient function-level simulations is correspondingly limited. By using conjugate symmetry we can extend the power of the HyperSim algorithm to functions that appear to be non-symmetric, and we can enhance the existing symmetries to create more efficient state machines than would be possible with conventional symmetries.

Although we have confined our attention to a single application, our on-going work suggests very strongly that the concepts introduced here will be useful in other areas of design automation such as three-level function minimization [34-37] and other areas.

II. Total, Partial and Weak Symmetry

Although the formal definition of symmetric functions appears in many other places, we will repeat the basic material here in the interest of making this paper self contained. Virtually any discussion of symmetry must begin with the concept of a permutation.

Definition 1. A permutation of degree *n* is a one-to-one function from a set *X* of *n* elements to itself.

The permutation formalizes the concept of "rearranging things." When we apply a permutation *p* to an element *a* of the set *X* to obtain *p(a)*, we can think of this operation as moving element a to position *p(a)* as shown in Fig. 1. It doesn't matter which set *X* we use, any set of size *n* will do. Many examples use the set of integers from 1 through *n*, but in our case *X* will be the set of inputs of an *n*-input Boolean function. We formally define this concept in Definition 2.

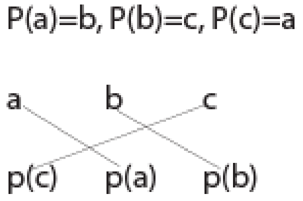

Fig. 1 Permutation Actions.

Definition 2. Let *p* be a permutation of degree *n* and let f be an *n*-input Boolean function. Let *p*(*f*) denote the function obtained by rearranging the inputs of *f* with permutation *p*. We say that *f* is *invariant* with respect to *p* if *f* and *p*(*f*) are the same function.

The set of all permutations forms an elementary algebraic structure called a group, which is defined formally in Appendix A. The function composition of two permutations p and q is called the *product* of p and q and is usually written pq rather than $q(p(x))$. The set of all permutations on a set of n elements is a group with respect to this product. There are many types of groups other than just permutation groups. For example, the set of all $n \times n$ non-singular matrices also forms a group with respect to matrix multiplication. In this paper we will be concerned permutation groups and matrix groups and the relationships between them. To do this we need the concept of mapping one group into another while preserving the algebraic structure. This is given in the following definition.

Definition 3. Let *h* be a function from the group *G* to the group *H*. *h* is said to be a *homomorphism* if it preserves the group multiplication in the sense that for all $a,b \in G$, $h(ab) = h(a)h(b)$. Note that the multiplication on the left is in *G*, while the multiplication on the right is in *H*. A homomorphism is called an *isomorphism* if it is one-to-one. Two groups *G* and *H* are said to be *isomorphic* if there is an isomorphism from *G* to *H* and an isomorphism from H to G. If two groups are isomorphic, they are essentially the same group with "different names" for the elements.

Groups can be used to characterize the symmetry of a Boolean function. The set of all permutations that leave a Boolean function f invariant forms a group. (The proof of this is elementary.) These observations allow us to make the following definition.

Definition 4. The set of permutations $G = \{p \mid f \text{ is invariant with respect to } p\}$ is a group called the *symmetry group of f*.

The symmetry group of a function *f* always exists, because the identity function is a permutation. The set of all permutations on a set of *n* elements is called the *symmetric group* of order n and is denoted S_n. The classes of symmetry mentioned above, total, partial, and weak, are characterized in terms of the symmetry group of a function. The following definitions formally define these classes of symmetry. (As stated in the introduction, in electronic design automation the terminology for these concepts varies widely from one author to another. Our terminology is in common use in mathematics and other fields.)

Definition 5. An *n*-input Boolean function *f* is said to be *non-symmetric* if the symmetry group of *f* contains only the identity element. Otherwise *f* is said to be *symmetric*.

Definition 6. An *n*-input Boolean function *f* is said to be *totally symmetric* if the symmetry group of *f* is S_n.

A function is partially symmetric if it has one or more subsets of variables that can be permuted in any fashion, as in the functions $abc + d$ and $ab + c + d$. We will omit the formal definition because it is surprisingly complicated and adds little or nothing to the understanding of the concept. The formal definition of weak symmetry is given in Definition 7, but in the remainder of this paper will be concerned only with total and partial symmetries.

Definition 7. An *n*-input Boolean function *f* is said to be *weakly symmetric* if it is symmetric but neither totally symmetric nor partially symmetric.

III. Functional Equivalences

Another important property of Boolean circuits is functional equivalence. Functional equivalences exist when the available signals are not independent of one another. These equivalences can be exploited to simplify both the design and layout of digital circuits. The classic example of the exploitation of functional equivalence is mixed-level design, in which the equivalence between a signal and its inverse is used to simplify the design process. Although inversions are the simplest type of functional equivalences, there are many others. Suppose we have three signals available, A, B, and $A\oplus B$, where $\oplus$ represents the exclusive-or function. It is possible to eliminate any one of these signals because any one of these signals can be computed from the other two. The choice of which signal to eliminate depends on the functions that need to be computed, and $A\oplus B$ is not automatically the best choice.

When functional equivalence is combined with symmetry, new types of symmetries arise. For example the function $f(a,b)=a'+b$ is symmetric with one input inverted with respect to the other. This is a combination of total symmetry and the inversion functional equivalence. This type of symmetry can be exploited in dual rail designs and in applications like the inversion algorithm[1] that are insensitive to inversions. We call this type of symmetry *skew symmetry*. (This term means something slightly different when dealing with real-valued functions.)

Formally, skew-symmetry is defined with respect to transformations of the form $N(i)$, which inverts input *i* of a function. Two or more of these transformations can be combined to invert several inputs. For example the transformation $N(i,j)$ inverts both input *i* and input *j*. We can use these transformations to give a formal definition of skew symmetry.

Definition 8. Let *I* be a set of inputs of the Boolean function *f*, and let *N*(*I*)(*f*) be the function obtained from *f* by inverting the inputs in the set *I*. The function *f* is said to be *skew-symmetric* if *N*(*I*)(*f*) is symmetric for some non-empty proper subset of inputs *I*.

Skew symmetry can be further refined into total skew-symmetry, partial skew-symmetry and weak skew-symmetry. The definitions of these terms are obvious. To exploit more complex functional equivalences, it is necessary to adopt a slightly different view of *n*-input Boolean functions. Rather than viewing an *n*-input Boolean function *f* as a function of *n* variables, we choose to view *f* as a function of a single vector-valued variable. Although this seems like splitting hairs, this change of viewpoint will allow us to exploit a rich body of mathematics that would be unavailable otherwise.

Given a set of *n*-element input vectors, K, any one-to-one mapping from K to itself defines a functionally equivalent set of input signals. Rather than using arbitrary one-to-one vector functions we will concentrate on non-singular linear transformations. The reasons for this are two-fold: linear transformations are easy to specify and their underlying theory is well understood.

To proceed from this point it is necessary to be explicit about what we mean by a vector and a linear transformation. First, it is necessary to specify the underlying algebra that we will be using for both vectors and linear transformations. Rather than use the usual Boolean algebra with its AND, OR and NOT operations, we will use an algebraic structure known as a *field*. This is necessary if we are to fully exploit the mathematics of linear transformations.

The most familiar fields are the real and complex numbers, but there are many others. In particular the integers modulo *p*, a prime number constitutes a field. The integers modulo 2 is a field which is designated GF(2) and contains only two elements 0, and 1. A field has two operations, addition and multiplication which must obey the same algebraic laws as real numbers (see Appendix A for a formal definition). In GF(2) multiplication corresponds to the familiar AND function, while addition corresponds to Exclusive-OR. Addition and subtraction are identical, greatly simplifying algebraic operations. The theory of error-correcting codes makes extensive use of the properties of GF(2), but we will exploit these properties in a different way to define new, more extensive classes of symmetric Boolean functions.

As with real numbers, the elements of an *n*-dimensional vector space over GF(2) can be written as ordered *n*-tuples of ones and zeros. Because GF(2) is finite, any finite dimensional vector space over GF(2) is also finite. As with real numbers it is GF(2) vectors can be linearly independent. The set of vectors (110), (011), and (001) is linearly independent, while the set (110), (011), and (101) is linearly dependent.

Linear transformations of GF(2) vector spaces can be represented as $n \times m$ matrices of ones and zeros. All of the familiar matrix algebra works exactly the same as for real numbers. A matrix is non-singular if it is one-to-one, which is equivalent to saying that its rows are all non-zero and linearly independent. Matrix multiplication works as usual, as does Gaussian elimination, the calculation of determinants and the extraction of characteristic polynomials. Eigen values are not always defined, but the characteristic polynomial always exists.

The main reason why it is necessary to consider linear transformations is because we wish to express our notions of symmetry in terms of permuting input vectors rather than permuting function inputs. It is not possible to use a permutation to rearrange the elements of a vector. While it is true that one could apply permutations to vectors in an intuitive way, technically speaking permutations are defined on sets of *n* distinct elements. However, permutations can be associated with a class of matrices in a natural way, and these matrices *can* be used to permute vectors. The matrices in question are the permutation matrices. They are defined as follows.

Definition 9. A permutation matrix is an $n \times n$ matrix whose entries are all zero except for a single 1 in each row and each column.

There are $n!$ $n \times n$ permutation matrices, and the set of all $n \times n$ permutation matrices forms a group that is isomorphic to the group S_n. For example, consider the set of all permutations on a set of 3 elements, $S_3 = \{123, 132, 213, 321, 312, 231\}$. The matrices corresponding to these six permutations are given in Fig. 2.

$$\begin{pmatrix}1&0&0\\0&1&0\\0&0&1\end{pmatrix}\begin{pmatrix}1&0&0\\0&0&1\\0&1&0\end{pmatrix}\begin{pmatrix}0&1&0\\1&0&0\\0&0&1\end{pmatrix}\begin{pmatrix}0&0&1\\0&1&0\\1&0&0\end{pmatrix}\begin{pmatrix}0&1&0\\0&0&1\\1&0&0\end{pmatrix}\begin{pmatrix}0&0&1\\1&0&0\\0&1&0\end{pmatrix}$$

Fig. 2. The Standard Representation of S_3.

The set of all $n \times n$ non-singular matrices over GF(2) forms a group called the General Linear Group of order *n* over GF(2) and is designated $GL_n(2)$. When we map a group into a group of non-singular linear transformations we call the result a *Group Representation*. We formalize this concept in the following definition.

Definition 10. Let *G* be any group, and let *M* be a homomorphism from *G* to $GL_n(F)$, where $GL_n(F)$ is the general linear group of order *n* over some field *F*. *M* is said to be a *representation* of the group *G*. If *M* is an isomorphism, then *M* is said to be *faithful*.

Although a representation is technically an isomorphism from a general group to a group of matrices, it is less confusing to refer to the matrix group itself as the representation, particularly since there are usually many different isomorphisms between the general group and the matrix group. The group of all of $n \times n$ permutation matrices over a field F is called the *standard representation* of S_n over *F*. The standard representation of S_n over GF(2) is designated $SR_n(2)$. The key point of this discussion is that the standard representation of S_n over GF(2) *is not the only representation of* S_n in $GL_n(2)$. Fig. 3 gives a second representation of S_3.

$$\begin{pmatrix}1&0&0\\0&1&0\\0&0&1\end{pmatrix}\begin{pmatrix}1&0&0\\1&1&1\\0&0&1\end{pmatrix}\begin{pmatrix}1&0&0\\1&0&1\\1&1&0\end{pmatrix}\begin{pmatrix}1&0&0\\0&1&0\\0&1&1\end{pmatrix}\begin{pmatrix}1&0&0\\1&1&1\\1&1&0\end{pmatrix}\begin{pmatrix}1&0&0\\1&0&1\\0&1&1\end{pmatrix}$$

Fig. 3. An Alternative Representation of S_3.

The representation of Fig. 3 is a *conjugate* of the standard representation. To understand what this means, we must define the concept of similar matrices. The definition for GF(2) matrices is the same as that for matrices of real numbers.

Definition 11. Two $n \times n$ matrices *M* and *N* are *similar* if there exists a non-singular $n \times n$ matrix T such that $M = T^{-1}NT$ where T^{-1} is the multiplicative inverse of *T*. The matrix *M* is said to be a *conjugate* of *N*, or the *conjugate of N with respect to T*.

The conjugate of a matrix group is obtained by finding the conjugate of every member of the group, as defined in Definition 12.

Definition 12. Let *G* be a group of $n\times n$ matrices and let *T* be a non-singular $n\times n$ matrix. The *conjugate of G with respect to T* is denoted $T^{-1}GT$ and is the set $T^{-1}GT=\{T^{-1}MT \mid M\in G\}$.

If *G* is a representation of S_n, then so is $T^{-1}GT$. Sometimes *G* and $T^{-1}GT$ are identical, but in general they are different. The two representations given in Fig. 2 and Fig. 3 are $SR_3(2)$ and the conjugate of $SR_3(2)$ with respect to the matrix of Fig. 4. There are alternative representations of S_n that are *not* conjugate to the standard representation, but at the present time it is not clear that these representations have any value.

$$\begin{pmatrix}1&1&0\\0&1&1\\0&0&1\end{pmatrix}$$

Fig. 4. A Symmetry Matrix.

IV. Conjugate Symmetry

The standard representation of S_n gives us an alternative way to define symmetric functions. We can treat an *n*-input Boolean function, f, as if it were a function defined on GF(2) vectors. Because a $n\times n$ matrix, M, is a function from vectors to vectors, we can find the composition of f and M, $f(M(v))$ which we usually write fM. Using these concepts, we can define symmetric functions in the following way.

Definition 13. A Boolean function f is symmetric if there is a matrix $M\in SR_n(2)$ with $M\neq I$ the identity matrix, such that $fM=f$.

Definition 13 is essentially identical to Definition 5, but it can be extended to non-standard representations of S_n. In particular, we can formulate our definition of conjugate symmetry in the same fashion.

Definition 14. A Boolean function f possesses *conjugate symmetry* if there exists a representation $R\subseteq GL_n(2)$ which is conjugate to $SR_n(2)$ and a matrix $M\in R$, with $M\neq I$, such that $fM=f$. The function f is said to be *C-symmetric with respect to* R. If every element of R leaves f invariant, then f is said to be *totally C-symmetric* with respect to R. A function can also be partially C-symmetric or weakly C-symmetric with respect to R.

Unlike ordinary symmetry, it is difficult to express conjugate symmetry in terms of function variables. Let SB_n be the set of all *n*-element Boolean vectors of weight 1, i.e. the set (1,0,0,…,0), (0,1,0,…,0), … . SB_n is a basis of the vector space of dimension n over $GF(2)$. Any *n*-element vector can be generated by bit-wise xoring elements of SB_n. Any vector of weight 2 can be obtained by xoring two elements of SB_n, any vector of weight 3 can be obtained by xoring 3 elements of SB_n, and so forth. Let $SB_n(k)$ be the set of all *n*-element vectors of weight *k*. If a function is totally symmetric, it must take the same value for every element of $SB_n(k)$. Although we usually speak of permuting variables or permuting the elements of input vectors, at the most fundamental level, it is actually the elements of SB_n that are being permuted with one another. Because there is an obvious one-to-one correspondence between the elements of SB_n and the input variables of a function, and because any permutation of an element of SB_n produces another element of SB_n, it is easy to understand ordinary total symmetry in terms of permutations of variables or permutations of input vectors. For conjugate symmetry this is not possible. Let A_n be any basis of

GF(2). The elements of A_n need to be linearly independent, but need not have any other relationship. Furthermore, there usually isn't any obvious correspondence between the elements of A_n and the input variables of a function f. Let $A_n(k)$ be the set of vectors that is obtained by xoring all combinations of the elements of A_n taken *k* at a time. For example, if $A_n = \{v_1, v_2, v_3\}$ then $A_n(2) = \{v_1 \oplus v_2, v_1 \oplus v_2, v_2 \oplus v_3\}$. $A_n(0)$ is just the zero vector. If a function f is C-symmetric, there is a basis A_n such that f has the same value on every element of $A_n(k)$ for all *k*. The function f permutes the elements of A_n. Because every element of A_n is a linear combination of the elements of SB_n, it may appear that f permutes linear combinations of its input variables. However, this is not the case, as we will show below.

There is a natural relationship between conjugate symmetries and symmetric Boolean functions. Let R be a representation of S_n in $GL_n(2)$ with $R = T^{-1}SR_n(2)T$. If f is a totally symmetric function, then fT is totally symmetric with respect to R. It is easy to show that this is the case. If $M \in R$ then $M = T^{-1}NT$ for some $N \in SR_n(2)$. We wish to show that M leaves fT, in other words that $fTM = fT$. This is easy to show since $fTM = fTT^{-1}NT$. Because TT^{-1} is the identity matrix, $fTT^{-1}NT = fNT$. Because f is totally symmetric and $N \in SR_n(2)$, $fN = f$, so $fNT = fT$. This implies that if g is C-symmetric, it be factored into a function f and a matrix M such that $g = fM$ and f is symmetric in the ordinary sense.

Suppose that $g(a,b,c)$ is totally C-symmetric with respect to the matrix of Fig. 4. Because any matrix can be implemented as a set of exclusive-or gates, g can be factored into $g(a,b,c) = f(a, a \oplus b, b \oplus c)$, where f is totally symmetric. (The xor expressions are obtained from the columns of the matrix.). Because f is totally symmetric, $g(a,b,c)$ is also equal to $f(a \oplus b, b \oplus c, a)$, and so forth. However, before we can permute the linear combinations of the inputs we must first find the function f. Fig. 5 gives the inverse of the matrix of Fig. 4. If we substitute this matrix into $g(a,b,c)$ as follows: $g(a, a \oplus b, a \oplus b \oplus c)$ we obtain f. Because f is totally symmetric, we can permute the variables of the linear combinations without changing f. Thus $f(a,b,c) = g(a, a \oplus b, a \oplus b \oplus c) = g(c, c \oplus b, c \oplus b \oplus a)$. However, because g is not totally symmetric in the usual sense, we cannot permute the linear combinations a, $a \oplus b$ and $a \oplus b \oplus c$ with one another.

$$\begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}$$

Fig. 5. A The Inverse of a Symmetry Matrix.

V. The HyperSim Simulator

Our interest in conjugate symmetries originally stemmed from the fact that one of our most powerful simulation algorithms [1] can be significantly more efficient if the function being simulated is symmetric. If the Boolean function, g, can be factored into $g = fT$ where f is symmetric, and the cost of the transformation T is low, then we have an opportunity to improve the performance of our simulations. The most straightforward way to implement T is to insert additional xor gates into the circuit, and replace the C-symmetric function g with its symmetric counterpart f. This is essentially what we do, but in our simulation engine we can eliminate a portion of the processing for the xor gates by routing events into f in a creative way. Our simulator, which is called HyperSim, is a compiled simulator. The circuit is translated into run-time code which is then compiled and executed to simulate the circuit. Symmetry detection is done during the code-generation phase, so the cost can be amortized over many simulations.

In HyperSim, a function simulation contains of three types of state machines, input state machines, gate state machines, and output state machines, as shown in Fig. 6. The *Input* state machines are used to process the inputs of a function. Typically they toggle between two states which correspond to the 1/0 state of the physical input. The *Gate* state machine is used to determine when event propagation occurs. The gate state machine may represent a conventional gate or a more complex function. It processes signals from several input state machines and signals the output state machine when an output transition occurs. The *Schedule* state machine is used to schedule events or to cancel previously scheduled events. Since two successive changes in a net cancel one another, the output machine toggles between the *scheduled* and *unscheduled* state. The gate state machine of Fig. 6 is used to simulate a totally symmetric function with three inputs.

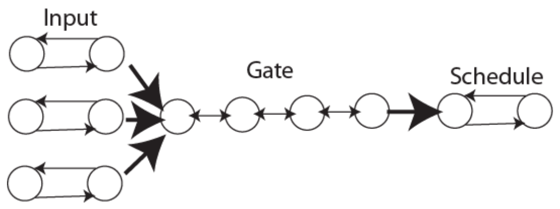

Fig.6. A Gate Simulation.

We have implemented these state machines in several different ways, but the simplest method is to incorporate all three of them into a single event-processing routine. For the gate machine of Fig. 6, the state is maintained as a single integer variable. Since all transitions are to neighboring states, each input event will either increment the gate state by one or decrement it by one. The actions of the event processor alternate with one another, so if an event on a particular input increments the gate state, the next event on that input will decrement it, and vice-versa. For the purposes of illustration, we will show the states of the net machine and scheduling machine as integer variables containing one or zero. The run time code has been optimized to remove most state variables, but the logic is identical to the pseudo-code given in Fig. 7. Event routing is determined statically during the code-generation phase of the simulator. Each primary input and each gate output is assigned a list of events, one per fan-out branch. The events are stored in pre-allocated pre-linked structures to permit fast scheduling and descheduling of the event list.

```
If NetState is 1
    Increment GateState
    NetState = 0
Else
    Decrement GateState
    NetState = 1
If  Gate Output Changes
    If OutputScheduled is 1
        DescheduleOutputEvents
        OutputScheduled = 0
    Else
        ScheduleOutputEvents
        OutputScheduled = 1
Go to next event
```

Fig. 7. Event Pseudo Code.

The in gate state machine of Fig. 6, each state represents all vectors of a certain weight, and a change in a single input will change the weight of the current input by one in either a positive or negative direction. Each state is also associated with an output value which is used to determine whether the gate-output has changed. For AND and OR gates, the computation is trivial and can be determined entirely from the state value, however for more complex functions each state has an associated value which is stored in a one-dimensional array. These values are used to determine whether the output of the function has changed.

If a function is not totally symmetric, the gate machine must be multi-dimensional. In the worst case it will have one dimension for each input variable and will be an n-dimensional hyper-cube. However partial symmetries allow states to be combined, reducing the number of dimensions. For example consider the three-dimensional structure pictured in Fig. 8. This represents a four input function with a partial symmetry in the last two variables. The X dimension represents the weight of the last two inputs, which can be 0, 1, or 2, and the Y and Z dimensions represent the first two variables which can have a “weight” of 0 or 1. The state of this gate machine is represented as three integer variables which give the coordinates of the current state. Separate event processing routines are used for each dimension and the correspondence between inputs and dimensions is fixed, so computing the new state is no more complex than for one-dimensional state machines. However determining whether the function output has changed requires finding a value in an n-dimensional array. Depending on how the lookup is implemented, this can require several instructions per dimension per event, so reducing the number of dimensions will reduce the computation time for each event.

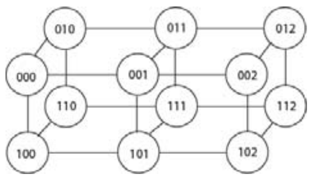

Fig. 8. A Hyperlinear Structure.

For ordinary symmetric functions, the inputs of the function and the input state machines are in one-to-one correspondence. When an event occurs on a function input, a single event is executed. For C-symmetric functions, we compute the symmetry matrix T by breaking the correspondence between input events and input state machines. An event on a function input may cause several events to be executed. When there is an event on more than one function input, the multiple scheduling may cause several events to be executed for a single input state machine. The input state machines function in such a way as to compute the exclusive or of two sequential events which has the effect of computing the symmetry matrix on the input vector. The symmetry matrix, T determines how events directed to input state machines. For example, Fig. 9 Fig.shows a 3-input Boolean function that is C-symmetric with respect to the matrix of Fig. 4. The function of Fig. 9 can be factored onto fT, where f is the totally symmetric function $a'b'c'+ab'c+a'bc+abc'$.

The columns of the matrix T determine how events are routed to the input state machines. Each column represents an input state machine, and each row represents a function variable. The simulation of f will have three input machines numbered 1, 2 and 3. Events from variable a will be routed to machine 1, events from variables a and b will be routed to input machine 2, and events from variables b and c will be routed to input machine 3.Fig. Fig. 10 shows how events are routed in an ordinary 3-input machine, and how they are routed using the symmetry matrix of Fig. 4. This scheme causes extra events to be generated for each input-change, but the cost of processing these events is offset by reducing the complexity of the event processing function. As with ordinary events, event routing is determined at code generation time. The extra events due to conjugate symmetry are simply added to the static list of events normally scheduled for a gate output or primary input.

a,b,c	Output
0,0,0	1
0,0,1	0
0,1,0	1
0,1,1	0
1,0,0	1
1,0,1	0
1,1,0	1
1,1,1	0

Fig. 9. A Function with Conjugate Symmetry.

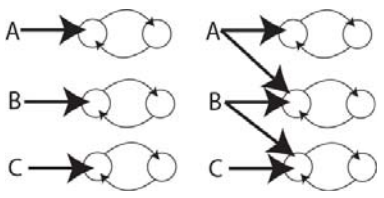

Fig. 10. Conjugate Symmetry Routing.

The main problem in detecting conjugate symmetry for a function g, is finding f and T such that $g = fT$. Our factorization technique is based on our technique for detecting symmetries, which uses *n*-dimensional lattices. We start with an *n*-dimensional hypercube whose vertices are labeled with the values of the function. By comparing the function values along various diagonals we are able to detect partial symmetries with respect to pairs of variables. Fig. 11 illustrates the detection of ordinary symmetry and skew symmetry for a two-input function. The vertices of the hypercubes are labeled with the input values. The function must have the same value for the two circled inputs for the corresponding symmetry to exist. By combining variables into "hypervariables" we are able to continue the process until all partial and total symmetries have been detected. Combining two variables produces non-cubic, hyperlinear structures such as the three-dimensional structure pictured in Fig. 8.

The reason why skew symmetry can be detected using the opposite diagonal is because inverting an input reflects the hypercube along the dimension corresponding to the inverted input. However, instead of comparing the opposite diagonal, our simulator reflects the cube along one dimension and checks for ordinary symmetry. (The state labels 00, 01, 10, and 11 are for information only. They do not exist in the data structure representing the hypercube.)

Reflecting the hypercube is much easier than it sounds. Because our algorithm changes the number of dimensions of the cube, it cannot be implemented as a standard data structure. Instead, we store all vertices in a linear array and use our own indexing function to access the vertices. This indexing function is identical to that used by compilers to access multi-dimensional arrays, with the exception that the number of dimensions is not fixed. To reflect the cube, we tag one dimension so that the indexing function indexes the dimension in reverse order. In Fig. 11, suppose that the vertical dimension is tagged for reverse indexing. The index (1,0) will access the node labeled 11 and the index (0,1) will access the node labeled 00. Thus testing for conventional symmetry will compare the vertices labeled 00, and 11 instead of 10 and 01.

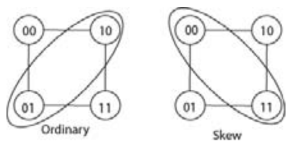

Fig.11. Symmetry Detection.

Our symmetry detection algorithm can be adapted to detect conjugate symmetries by observing the effect of a symmetry matrix on the function lattice. Consider the matrix of Fig. 12, which has a main diagonal of all 1's, a single 1 off the main diagonal, and zeros elsewhere. (This is an example of a *single-bit matrix* which we will discuss in more depth later.)

$$\begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Fig. 12. A Single-Bit Matrix.

Suppose f is a four-input function with inputs *a*, *b*, *c*, and *d*, and that f is symmetric with respect to the matrix of Fig. 12. This matrix replaces the input c with exclusive-or of a and c. Viewed in another way, the variable *c* is *conditionally inverted* by the variable *a*. That is, the variable c is inverted only if the variable a is equal to 1. A conditional inversion reflects the odd-numbered sub-planes of the hypercube along the dimension corresponding to the conditionally inverted input. The even numbered planes are left intact. (We number the planes starting with zero.) To check for a conjugate symmetry with respect to a single-bit matrix, indexing the odd-numbered planes in reverse order and the even-numbered planes in forward order will compensate for the conditional inversion. In our indexing algorithm we tag two dimensions, an inverted dimension and an inverting dimension. The indexing algorithm processes the specified coordinates of a vertex one at a time. When the coordinate for the inverted dimension is processed, the coordinate for the inverting dimension is tested, and if it is an odd number, the inverted dimension calculation is done in reverse order. As with skew symmetry, we tag the appropriate dimensions for the indexer and then test for ordinary symmetry.

Conjugate symmetries are detected with respect to pairs of variables. For a pair of variables, *a*, and *b*, it is necessary to do two separate checks, because *a* can be conditionally inverted by *b*, and *b* can be conditionally inverted by *a*. It is technically possible for a and b to conditionally invert one another, but for simple variables conditionally inverting both dimensions results in a test for ordinary symmetry. In Fig. 11, the two possible conditional inversions correspond to reversing the bottom row or the right-most column.

When pair of symmetric variables is detected, the dimensions that correspond to them are combined. This has the effect of combining the two into a single variable. The vertices along each diagonal are combined into a new structure and the unaffected vertices are copied intact into the new structure. The indexing tags that were used for the symmetry detection are left set during the creation of the new hyperlinear structure. This has the effect of undoing the reversals caused by skew and conjugate symmetries. For skew symmetries we record flag the inverted input, and for conjugate symmetries, we record the single-bit matrix that represents the conditional inversion.

When testing for symmetries in structures that have combined variables, several diagonals need to be tested. Fig. 13 shows the diagonals that will be tested in a 3×2 structure. The two vertices along a single diagonal must have the same value, but the two diagonals may have different values. When there are more than two dimensions, it is necessary to repeat each test for each plane. For the structure of Fig. 7 we must test the two diagonals in the forward plane and the two diagonals in the back plane. The skew and conjugate symmetry indexing mechanisms work the same for these types of structures as they do for square structures.

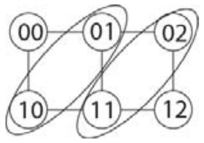

Fig. 5. A Non-Square Structure.

When testing structures with combined variables, it is possible for either the inverting variable or the inverted variable to be a combined variable. When this occurs more than one single-bit matrix is needed to record the reversal. Suppose the combined variable $(a_1,...,a_n)$ is conditionally inverted by the combined variable $(b_1,...,b_m)$. In this case $m \times n$ single-bit matrices are required, with each of the b_i inverting each of the a_j. As shown below, the product of these matrices will have ones on the main diagonal and $m \times n$ ones off the main diagonal in the positions corresponding to the ones in the single-bit matrices. Thus, we can take a shortcut and specify the combined matrix directly.

Once all symmetries have been found, we multiply all of the individual matrices together in the order they were found. The resultant hyperlinear structure is used as the gate state machine of the function *f* and the product matrix is the symmetry matrix T. After creating the input state machines and routing the input events according to the columns of the symmetry matrix, T, the resultant simulation structure will simulate the original function $g = fT$.

To reduce the complexity of the algorithm we start with a representation of the function in a zero-dimensional structure, and compute the cofactors with respect to successive variables. Instead of computing the entire n-dimensional hypercube we compare cofactors and collapse the structure as soon as any symmetries are found. This procedure is usually more efficient than starting with the complete hypercube. Also, to save space we compute the symmetry matrix on the fly rather than saving each single-bit matrix. The pseudo-code for the symmetry detection algorithm is given in Fig. 14. In this algorithm we assume that the inputs of the function are numbered from 0 to $n-1$. This pseudo-code has two subroutines, one for detecting ordinary symmetry and one for collapsing dimensions x and y. These algorithms are given in Fig. 15 and Fig. 16.

```
Create a zero dimensional structure H and place the $f$ in the single node.
Compute $f_0$ and $f_1$ the cofactors of $f$ with respect to input zero.
Replace H with a one dimensional structure containing $f_0$ in node zero and $f_1$ in node 1.
For v = each input variable from 1 through $n-1$
    For each node in H
        $g$ = the function contained in H
        Compute the cofactors $g_0$ and $g_1$ of $g$
    End For
    $d$ = The dimension of H.
    Replace H with a $d+1$ dimensional structure with the 0 cofactors first and the 1 cofactors following.
    // enumerate each pair of dimensions
    For x = 0 through $d$
        For y = x+1 through $d+1$
            Search for ordinary symmetry between dimensions x and y.
            If no symmetry found
                Tag dimension x for skew symmetry.
                Search for ordinary symmetry between dimensions x and y
                Un-tag dimension x.
            End If
            If no symmetry found
                Tag dimension x as conditionally inverting y
                Search for ordinary symmetry between dimensions x and y
                Un-tag dimensions x and y.
            End If
            If no symmetry found
                Tag dimension y as conditionally inverting x
                Search for ordinary symmetry between dimensions x and y
                Un-tag dimensions x and y.
            End If
            If no symmetry found
                Tag dimension x for skew symmetry
                Tag dimension y as conditionally inverting x
                Search for ordinary symmetry between dimensions x and y
                Un-tag dimensions x and y.
            End If
            If no symmetry found
                Tag dimension x for skew symmetry
                Tag dimension y as conditionally inverting x
                Search for ordinary symmetry between dimensions x and y
                Un-tag dimensions x and y.
            End If
            If symmetry found
                Re-tag x and y if necessary.
                Collapse x and y.
                Replace H with a $d$ dimensional structure containing the collapsed nodes.
                Un-tag x and y if necessary.
            End If
        End For
    End For
```

Fig. 14. Pseudo Code for Symmetry Detection.

```
$d$ = the dimension of H.
XLen = Length of dimension x
YLen = Length of dimension y
Create a $d$-element index vector IX
Create a $d-2$-element enumeration vector E
For each $d-2$-element vector E            ($2^{n-2}$ vectors)
    Distribute the elements of E into IX, skipping positions x and y.
    // check the diagonals that begin at the top of the structure
    For Pos = 1 through XLen-1
        IX[x] = Pos; IX[y] =0;
        TestVal = H[IX]
        IX[x] = IX[x] – 1
        IX[y] = IX[y] + 1
        While IX[x] >= 0
            If H[IX] is not equal to TestVal
                Return NOT SYMMETRIC
            End If
            IX[x] = IX[x] – 1
            IX[y] = IX[y] + 1
        End While
    End For
    // check the diagonals that begin at the right side of the structure
    For Pos = 1 through YLen-2
        IX[x] = XLen-1; IX[y] =Pos;
        TestVal = H[IX]
        IX[x] = IX[x] – 1
        IX[y] = IX[y] + 1
        While IX[y] < YLen
            If H[IX] is not equal to TestVal
                Return NOT SYMMETRIC
            End If
            IX[x] = IX[x] – 1
            IX[y] = IX[y] + 1
        End While
    End For
    Return SYMMETRIC
```
Fig. 15. Pseudo Code for Testing Variable Pairs.

```
d = the dimension of H.
XLen = Length of dimension x
YLen = Length of dimension y
NewLen = XLen + YLen – 1
Create a d-element index vector IX
Create a $d-1$ element index vector IY
Create a $d-2$ -element enumeration vector E
Create a $d-1$ dimensional structure K. The dimension y will be missing and the length of x will be NewLen.
        Otherwise the dimensions of K are the same length as the corresponding dimensions in H after accounting
        For the missing dimension.
For each $d-2$ -element vector E              ( $2^{n-2}$ vectors)
        Distribute the elements of E into IX, skipping positions x and y.
        Distribute the elements of E into IY skipping position x.
        IX[x] = 0
        IX[y] = 0
        IY[x] = 0
        K[IX] = H[IX]
        // Move the head of each diagonal into the new structure
        For Pos = 1 through XLen-1
                IY[x] = IY[x]+1
                IX[x] = Pos; IX[y] =0;
                K[IY] = H[IX]
        End For
        // now the diagonals that begin at the right side of the structure
        For Pos = 1 through YLen-2
                IY[x] = IY[x]+1
                IX[x] = XLen-1; IX[y] =Pos;
                K[IY] = H[IX]
        End For
        IY[x] = IY[x]+1
        IX[x] = XLen-1
        IX[y] = YLen-1
        K[IX] = H[IY]
        Replace H with K
```
Fig. 16. Pseudo Code for Collapsing a Dimension.

VI. An Example

Consider the function $g(a,b,c,d) = ac' + bc' + bd' + ab'd + a'cd'$, which has no ordinary or skew symmetries. First we eliminate variable a by setting $a = 0$ and then setting $a = 1$. The two resultant expressions are placed in a one-dimensional hypercube as shown in Fig 17.

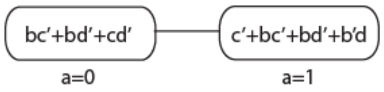

Fig. 6. Eliminating a.

Next we eliminate the variable b in the same way and create the two dimensional hypercube of Fig. 18.

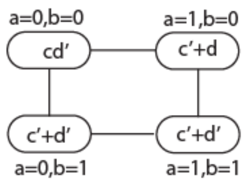

Fig. 78. Eliminating a and b.

If we reverse the right column of Fig. 18, the diagram will contain an ordinary symmetry. This implies that a and b are symmetric with a conditionally inverting b. The matrix T_1 of Fig. 19 defines this relationship.

$$T_1 = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Fig. 89. The First Symmetry Matrix.

We swap the two equations in the right hand column, and combine the two states containing $c' + d'$ giving the hyperlinear structure of Fig 20. We also save the matrix T_1 for future use.

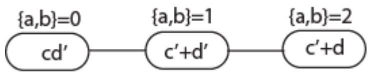

Fig. 20. Collapsing the State Machine 1.

We eliminate the variable c, giving the structure of Fig. 21.

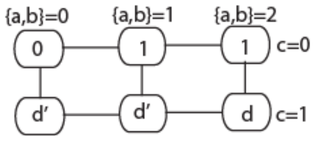

Fig. 21. Eliminating c.

This diagram will exhibit an ordinary symmetry if we reverse the center column. This corresponds to the variable c being conditionally inverted by the combined $\{a,b\}$. The inverted structure is shown in Fig. 22.

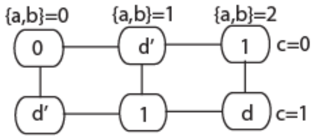

Fig.22. The Second Transformation.

Because $\{a,b\}$ is not a simple variable, we do not use a single-bit matrix to represent the conditional inversion. Instead we use matrix T_2 of Fig. 23, which is the product of two single-bit matrices. We save the matrix T_2 and combine the states containing 1 and the states containing d' to obtain the hyperlinear structure of Fig. 23.

$$T_2 = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Fig.23. The Second Symmetry Matrix.

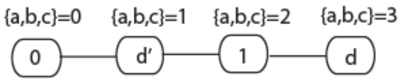

Fig.24. The Second Collapse.

Eliminating the final variable d gives us the structure of Fig. 25Fig..

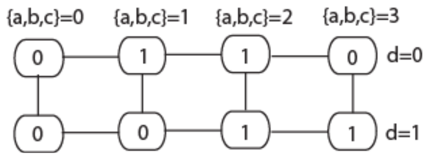

Fig.25. Eliminating d.

We invert columns 1 and 3, save the matrix of Fig. 26, and collapse the diagram to obtain the final state machine of Fig. 27.

$$T_3 = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Fig. 26. The Third Symmetry Matrix.

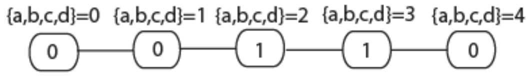

Fig. 27. The Final State Machine.

The final symmetry matrix is computed from T_1, T_2, and T_3 as shown in Fig. 28.

$$T_1 \times T_2 \times T_3 = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Fig. 28. The Final Symmetry Matrix.

VII. Single-bit Matrices

The previous section made extensive use of single-bit matrices. This approach may seem restrictive but any non-singular matrix can be represented as a product of single-bit matrices. Thus, we have not restricted ourselves by taking this approach. It turns out that single-bit matrices are useful in other algorithms as well, so the results of this section are widely applicable. It is well known that any non-singular real matrix can be factored into a product of elementary matrices. These are matrices of the forms given in Fig 29, where m ranges over the real numbers. In these matrices, all unspecified main-diagonal elements are one and all other unspecified elements are zero.

$$Type1: \begin{pmatrix} 1 &&&&&& \\ & \ddots &&&&& \\ && 1 &&&& \\ &&& m &&& \\ &&&& 1 && \\ &&&&& \ddots & \\ &&&&&& 1 \end{pmatrix} \quad Type2: \begin{pmatrix} 1 &&&&&& \\ & \ddots &&&&& \\ && 1 &&&& \\ &&& \ddots &&& \\ && -m && 1 && \\ &&&&& \ddots & \\ &&&&&& 1 \end{pmatrix} \quad Type3: \begin{pmatrix} 1 &&&&&& \\ & \ddots &&&&& \\ && 0 && 1 && \\ &&& \ddots &&& \\ && 1 && 0 && \\ &&&&& \ddots & \\ &&&&&& 1 \end{pmatrix}$$

Fig. 29. Elementary Matrices.

In GF(2) the constants m and $-m$ of Fig. 29 must be 1, so matrices of the first type become the identity matrix, and matrices of the second type become single-bit matrices. Although it is not immediately obvious, matrices of the third type can be expressed as products of three single-bit matrices, as shown in Fig. 30. The proof of the GF(2) decomposition is essentially the same as that for real-valued matrices.

$$\begin{pmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & 1 & & 1 & & \\ & & & \ddots & & & \\ & & 0 & & 1 & & \\ & & & & & \ddots & \\ & & & & & & 1 \end{pmatrix}\begin{pmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & 1 & & 0 & & \\ & & & \ddots & & & \\ & & 1 & & 1 & & \\ & & & & & \ddots & \\ & & & & & & 1 \end{pmatrix}\begin{pmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & 1 & & 1 & & \\ & & & \ddots & & & \\ & & 0 & & 1 & & \\ & & & & & \ddots & \\ & & & & & & 1 \end{pmatrix}=\begin{pmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & 0 & & 1 & & \\ & & & \ddots & & & \\ & & 1 & & 0 & & \\ & & & & & \ddots & \\ & & & & & & 1 \end{pmatrix}$$

Fig. 30. Matrices of Type 3.

VIII. Experimental Data

The simulation algorithms described here have been implemented into the HyperSim package[38]. This package has been used to simulate many different circuits, including the ISCAS-85 benchmarks[39]. Every circuit we tested contained some sort of conjugate symmetry, which is not surprising since even the simple circuit of Fig 31 exhibits a conjugate symmetry. Under ordinary symmetry it is partially symmetric in the variables A and B, while under conjugate symmetry it is totally symmetric.

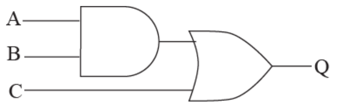

Fig. 31. A Simple Circuit with Conjugate Symmetry.

We compared the performance of the enhanced HyperSim[38] algorithm with the Inversion Algorithm[40], the fastest event-driven simulation available to us. The Inversion Algorithm is significantly faster than conventional even-driven simulation, and is competitive with levelized-compiled-code simulation at relatively high activity rates. HyperSim was able to detect many conjugate symmetries in each of the ISCAS-85 benchmarks, including C17. As Fig. 32 shows, the HyperSim gives a significant performance improvement over the Inversion Algorithm for most of these benchmark circuits. The numbers in Fig. 32 are seconds of CPU time for simulating 500,000 input vectors, not counting the time required to read input vectors and print output vectors. The timings were obtained on a 3.06 Ghz Xeon processor with 2GB of 233 Mhz memory.

Because the HyperSim simulator is designed to handle functions rather than entire circuits, its performance is dependent on the partitioning of gate-level circuits. The current implementation of the simulator partitions circuits into fanout-free networks, all of which have a single output. Partitions with more than eight inputs are broken into smaller networks. The input limit can be varied from 1 to 16 to obtain the best partitioning. For the benchmark circuits, we appear to detect the most symmetries when the input limit is set to a value between 8 and 12 inclusive. Any gates that are not part of a fanout-free network are simulated as individual gates. For the ISCAS-85 benchmark circuits, the majority of the gates will be simulated as individual gates, which limits the HyperSim's ability to optimize the simulation. However, there are a sufficient number of large partitions to permit an analysis of the symmetry detection algorithm on these circuits.

Both HyperSim and the Inversion Algorithm simulator are compiled simulators. The circuit is translated into an executable program which is then compiled and run to simulate the circuit. Symmetry detection is done during the translation phase, not during the run phase, so Fig. 32 does not include the time required to detect symmetries. However, despite the apparent complexity of the algorithm, symmetry detection is extremely fast. We tested each circuit for with input limits of 4, 8, 12 and 16. In all but one case symmetry detection took less than a second. The exception is circuit c3540 which took about four seconds with input limit 16.

Circuit	HyperSim	Inversion	Reduction
c432	1.336	2.068	35%
c499	1.974	2.336	15%
c880	2.782	4.672	40%
c1355	4.070	6.044	32%
c1908	4.066	8.922	54%
c2670	9.436	14.822	36%
c3540	9.506	15.928	40%
c5315	28.544	29.454	3%
c6288	22.766	31.378	27%
c7552	26.614	41.942	36%

Fig. 32. Experimental Data.

Fig. 33 gives a count of the various types of symmetry detected in each benchmark circuit. On the whole, the larger circuits appear to have more conjugate symmetries than the smaller circuits, which is not surprising. Every circuit exhibits at least some conjugate symmetries.

Circuit	Ordinary	Skew	Conjugate	SkewConj	Total
c432	47	45	9	2	103
c499	110	24	40	0	174
c880	133	4	13	2	152
c1355	14	24	0	104	142
c1908	138	4	4	0	146
c2670	279	10	11	58	358
c3540	211	0	232	0	433
c5315	298	12	120	239	669
c6288	0	0	480	0	480
c7552	710	16	108	119	953

Fig. 33. Symmetry Counts.

Fig. 34 shows the effect of partitioning on the number of circuit elements. Although increasing the input limit beyond 8 improves things somewhat, the increase in the limit does not substantially improve the partitioning.

Limit	4		8		12		16	
Circuit	Gates	FFNs	Gates	FFNs	Gates	FFNs	Gates	FFNs
c432	89	31	32	40	32	40	31	40
c499	32	58	0	58	0	58	0	58
c880	133	78	80	78	59	67	44	61
c1355	168	106	144	114	144	114	144	114
c1908	287	115	281	117	275	119	267	113
c2670	538	185	395	198	384	180	379	173
c3540	668	330	512	310	488	302	468	187
c5315	1021	290	565	334	470	339	339	467
c6288	976	480	976	480	976	480	976	480
c7552	1136	555	871	579	767	587	758	575

Fig. 34. The Effect of the Input Limit on Partitioning.

Fig. 35 shows the effect on simulation performance for the larger circuits, those with the most conjugate symmetries. These results are in CPU seconds of execution time. These results were obtained from a version of HyperSim that is not capable of measuring the time required for reading vectors and printing results. The results of Fig. 35 will be slightly higher than those of Fig. 32, because the vector read and print times are included in the results of Fig. 35.

For the most part these times show an improvement when symmetry is detected. The notable exception is circuit c6288. This circuit shows a serious degradation in performance when symmetry is detected. This is due to the fact that there are 480 multi-gate partitions in the circuit, each one of which has a single conjugate symmetry. This conjugate symmetry will cause extra events to be scheduled for each of the 480 partitions. Processing each event will be more efficient than for the circuit with no symmetry detection, but not efficient enough to give any performance gain. For there to be performance improvements, it is necessary to detect two or three conjugate symmetries per partition.

Circuit	No Sym.	Sym.
c2670	18.800	17.010
c3540	11.300	10.940
c5315	43.370	33.210
c6288	18.410	25.560
c7552	43.590	32.790

Fig. 35. The Effect of Symmetry Detection.

The symmetry detection is not particularly sensitive to variable ordering because all variable pairs are retested every time a new variable is added to the hyperlinear structure. However, it is sensitive to the order in which symmetries of different types are detected. We have endeavored to preferentially detect ordinary symmetries because these make event processing more efficient without adding new events. This effort has not been completely successful as Fig. 36 shows. This table shows the number of ordinary symmetries that are detected when conjugate symmetries are also detected and when conjugate symmetry detection is suppressed. In most cases the difference is small, but for circuit c6288, the difference is substantial, and profoundly affects simulation performance. More research is needed to determine how to deal with cases like this.

Circuit	With Conj.	Without
c432	47	63
c499	110	134
c880	133	138
c1355	14	134
c1908	138	138
c2670	279	295
c3540	211	225
c5315	298	300
c6288	0	464
c7552	710	736

Fig. 36. Ordinary Symmetries without Conjugate Symmetry Detection.

IX. Conclusion

We have shown how GF(2) matrices can be used to simplify the simulation of Boolean functions. This work greatly extends the number of functions that can be simulated efficiently, because the number of conjugate symmetries is quite large compared to the number of ordinary symmetries. For example, there are 65,536 4-input Boolean functions, 32 of which are totally symmetric. Adding skew symmetries helps, but it expands this total by at most a factor of 4. There are 420 classes of conjugate symmetry in $GL_4(2)$, giving us many, many more totally symmetric functions. When the number of partial symmetries is considered the total becomes much larger. The number of conjugacy classes increases with the number of inputs, so as the number of inputs increases, so does the opportunity to detect conjugate symmetries.

Although we have shown that conjugate symmetry is a powerful tool for enhancing gate-level simulations, the work that we have done barely scratches the surface. It is our belief that GF(2) matrices will eventually find many applications in diverse areas of EDA, and perhaps in other disciplines as well. We are currently pursuing research in the area of both logic minimization and symmetry breaking for SAT solvers.

Of course, one drawback of using GF(2) matrices is the cost of computing them. Usually the benefits outweigh the cost, but even so, if they were used early in the design cycle, it might be possible to get them for free. Imagine that a matrix is being used to transform a set of inputs. It may be possible to encode data differently from the outset, avoiding the cost of the matrix entirely.

One area that has a great deal of potential is the area of exotic symmetries. Exotic symmetries are those that go beyond conjugate symmetry. For example, in $GL_n(2)$ there are nine super-classes of S_4 representations. Conjugate symmetry constitutes only one of these super-classes. What about the remaining eight classes? The study of exotic symmetries is hampered somewhat by the fact that there are unsolved mathematical problems in the area. For example, there is, as yet, no method for enumerating all matrix groups isomorphic to S_n for an arbitrary n other than by brute force. The number of $n \times n$ non-singular matrices is given by the following formula.

$$\prod_{i=0}^{n-1}(2^n - 2^i)$$

This formula grows faster than exponential, making brute force analysis impractical for larger matrices. There are around 20 thousand non-singular 4×4 matrices. For 5×5 matrices the total is around 9 million and for 6×6 matrices the total is over 20 billion.

It is undoubtedly true that there are many applications of conjugate symmetry and GF(2) matrices that have not occurred to us. It is our firmest hope that this paper will inspire much more work in this area.

X. Appendix A, Formal Definitions.

Definition A1. Given a set *G* with a multiplication operation, *G* is a *group* if the following four conditions are met.

1 For any two elements $a \in G$ and $b \in G$, $ab \in G$. That is, the multiply operation is *closed*.

2 The multiplication operation is associative, that is, $a(bc) = (ab)c$ for all $a,b,c \in G$.

3 There is an identity element $i \in G$ such that for all $a \in G$ $ai = ia = a$.

4 For all $a \in G$ there is an inverse element a^{-1}, such that $aa^{-1} = a^{-1}a = i$.

Definition A2. A field is set *F* with two operations, addition and multiplication that have the following properties.

1 For all $a,b \in F$, $ab \in F$ and $a+b \in F$.

2 For all $a,b \in F$, $ab = ba$ and $a+b = b+a$.

3. For all $a,b,c \in F$, $a(bc) = (ab)c$ and $a+(b+c) = (a+b)+c$

4. For all $a,b,c \in F$, $a(b+c) = ab+ac$

5. There exists an element 0 such that $0+a = a+0 = a$ for all $a \in F$

6. There exists an element 1 such that $1a = a1 = a$ for all $a \in F$

7. For all $a \in F$, there exists $-a$ such that $a+(-a) = (-a)+a = 0$

8. For all $a \in F$ except 0, there exists a^{-1} such that $aa^{-1} = a^{-1}a = 1$.

Definition A3. A vector space over a field *F* is a set *V* with two operations called vector addition and scalar multiplication. These operations must obey the following laws.

1 For all $v,w \in V$, $v+w \in V$, and $v+w = w+v$.

2 For all $v,w,x \in V$ $v+(w+x) = (w+v)+x$

3.There is a vector 0 such that $0+v = v+0 = v$ for all $v \in V$.

4 For all $v \in V$ there exists a $-v$ such that $v+(-v) = (-v)+v = 0$.

5.For all $v \in V$ and $\alpha \in F$, $\alpha v \in V$.

7 For all $v,w \in V$ and $\alpha,\beta \in F$, $\alpha(\beta v) = (\alpha\beta)v$, $(\alpha+\beta)v = \alpha v + \beta v$, and $\alpha(v+w) = \alpha v + \alpha w$.

8.For all $v \in V$, $1v = v$, and $0v = 0$.

Definition A4. Given two vector spaces V and W over a field F, a linear transformation T from V to W is a function from V to W with the following properties.

1. For all $v,w \in V$, $T(v+w) = T(v)+T(w)$

2. For all $v \in V$ and $\alpha \in F$, $T(\alpha v) = \alpha T(v)$.

XI. References

1. Maurer, P., “Efficient Event-Driven Simulation by Exploiting the Output Observability of Gate Clusters,” *IEEE Transactions on CAD*, Vol. 22, No. 11, Nov., 2003, pp 1471-1486.
2. Burrow, M., *Representation theory of finite groups*, Academic Press, New York, 1965.
3. McWeeny, R., *Symmetry: An introduction to Group Theory and its Applications*, The International Encyclopedia of Physical Chemistry and Chemical Physics Topic 1, Volume 3, The MacMillan Company, New York, 1963.
4. Maurer, P,. “Using Conjugate Symmetry to Improve Simulation Performance,” *Design Automation and Test in Europe Conference*, Mar 2006.
5. Juringius, N, *Recherches sur les fonctions symétriques*, Imprimerie Håkan Ohlsson, Lund, 1931.
6. C. E. Shannon, A Symbolic Analysis of Relay and Switching Circuits, *AIEE Transactions*, Vol. 57, pp. 713-723, 1938.
7. Maurer, P., Schapira, A., A Logic-to-Logic Comparator for VLSI Layout Verification *IEEE Transactions on CAD*, Vol. 7, No. 8, Aug., 1988, pp 897-907.
8. Maurer, P., *An Application of Group Theory to the Analysis of Symmetric Gates*, Originally issued as a Bell Labs Internal Memorandum, April, 1986, reissued by the author as a Baylor University Technical Report http://hdl.handle.net/2104/5438, Nov. 2009.
9. Mohnke, J., Molitor, P. Malik, S, “Limits of Using Signatures for Permutation Independent Boolean Comparison,” *Formal Methods in System Design* 21(2): 167-191 (2002).
10. Kravets, V., Sakallah, K., “Generalized Symmetries in Boolean Functions,” *2000 International Conference on Computer-Aided Design (ICCAD '00)*, p. 526.
11. Maurer, P., The Complexity of Detecting Symmetric Functions, *Baylor Department of Computer Science Technical Report*, http://hdl.handle.net/2104/5267, Feb 2009.
12. Tsai, C., Marek-Sadowska, M., “Generalized Reed-Muller forms as a tool to detect symmetries,” *IEEE Trans. Comput.*, vol. 45, pp. 33-40, Jan. 1996.
13. Drechsler, R., and Becker, B., “Sympathy: fast exact minimization of fixed polarity Reed-Muller expressions for symmetric functions,” *1995 European Design and Test Conference* p. 91
14. C.C. Tsai and M. Marek-Sadowska. “Detecting symmetric variables in boolean functions using generalized Reed-Muller forms.” *International Symposium on Circuits and Systems*, 1994.
15. Maurer, P., “The Inversion Algorithm for Digital Simulation” *IEEE Transactions on Computer Aided Design*, Vol. 16, No. 7, July 1997, pp. 762-769.
16. Ulrich, E., “Event Manipulation for Discrete Simulations Requiring Large Numbers of Events,” *JACM*, V.21, N.9, Sep. 1978, pp. 777-85.
17. Szygenda, S., Rouse, D., Thompson, E., “A Model and Implementation of a Universal Time-Delay Simulator for Large Digital Nets,” *Spring Joint Computer Conference*, 1970, pp. 491-496.
18. Chiang, M., Palkovic, R., “LCC Simulators Speed Development of Synchronous Hardware,” *Computer Design*, Mar. 1, 1986, pp. 87-91.
19. Smith, P., Mercer, M., Brock, B., “Demand Driven Simulation: BACKSIM,” *Proceedings of the 24th Design Automation Conference*, 1987, pp.181-87.
20. Appel, A., “Simulating Digital Circuits with One Bit Per Wire,” *IEEE Transactions on Computer Aided Design*, Vol. CAD-7, pp. 987-993, Sept., 1988.
21. Lewis, D., “A Hierarchical Compiled Code Event-Driven Logic Simulator,” *IEEE Transactions on Computer Aided Design*, Vol 10, No. 6, pp.726-737, June 1991.
22. Olukotun, K., Heinrich, M., Ofelt, D., “Digital system simulation: methodologies and examples, *Proceedings of the 35th conference on Design automation*, 1998, pp. 658-663.
23. Luo, Y., Wongsonegoro, T., Aziz, A., “Hybrid Techniques for Fast Functional Simulation” *Proceedings of the 35th conference on Design automation*, 1998, pp. 664–667.
24. Ganai, M., Aziz, A., Kuehlmann, A., “Enhancing simulation with BDDs and ATPG,” *Proceedings of the 36th conference on Design automation*, 1999, pp. 385-390.
25. Wilson, C., Dill, D., “Reliable Verification Using Symbolic Simulation with Scalar Values,” *Proceedings of the 37th conference on Design automation*, 2000, pp. 124-129.
26. Kölbl, A., Kukula, J., Damiano, R., “Symbolic RTL simulation,” *Proceedings of the 38th conference on Design automation*, 2001, pp. 47-52.
27. Cadambi, S., Mulpuri, C., Ashar, P., “A Fast, Inexpensive and Scalable Hardware Acceleration Technique for Functional Simulation,” *Proceedings of the 39th conference on Design automation*, 2002, pp. 570-575.

28. Schubert, K. “Improvements in Functional Simulation Addressing Challenges in Large, Distributed Industry Projects,” *Proceedings of the 40th conference on Design Automation*, 2003, pp. 11-14.
29. Schilp, W., P. M. Maurer, “Unit Delay Simulation with the Inversion Algorithm,” *Proceedings of ICCAD-96*, pp. 412-7.
30. Schuler, D., “Simulation of NAND Logic,” *Proceedings of COMPCON 72*, Sept 1972, pp. 243-5.
31. Heydemann, M., Dure, D., “The Logic Automation Approach to Accurate Gate and Functional Level Simulation,” *Proceedings of ICCAD-88*, pp. 250-253.
32. Sosic, R, Gu, J., Johnson, R., The Unison Algorithm, Fast Evaluation of Boolean Expressions, *TODAES* Vol 1, No. 4, Oct. 1996, pp. 456-477.
33. Maurer, P., “Event Driven Simulation Without Loops or Conditionals,” *Proceedings of ICCAD-2000*, 2000, pp. 23-26.
34. Bernasconi, A., Ciriani, V., Fabrizio Luccio, F., Pagli L., “Three-Level Logic Minimization Based on Function Regularities,” I*EEE Transactions on Computer Aided Design*, Vol. 22, No. 8, Aug 2003, pp. 1005-1016.
35. Bernasconi, A., Ciriani, V., Luccio, F., Pagli, L., “Synthesis of Autosymmetric Functions in a New Three-Level Form,” *Theory Comput. Syst.* 42(4): 450-464 (2008)
36. Bernasconi, A., Ciriani, V., Luccio, F., Pagli, L., “Exploiting Regularities for Boolean Function Synthesis.” *Theory Comput. Syst*. 39(4): 485-501 (2006)
37. Bernasconi, A., Ciriani, V., DRedSOP: “Synthesis of a New Class of Regular Functions,” *Proceedings of the 9th EUROMICRO Conference on Digital System Design*, pp. 377-384.
38. Maurer, P., HyperSim, *Baylor Department of Computer Science Technical Report*, http://hdl.handle.net/2104/5489, Nov. 2009.
39. Brglez, F., P. Pownall, R. Hum, “Accelerated ATPG and Fault Grading via Testability Analysis,” *Proceedings of the International Conference on Circuits and Systems*, 1985, pp. 695-698.
40. Maurer, P., The Inversion-Algorithm Software, *Baylor Department of Computer Science Technical Report*, http://hdl.handle.net/2104/5487, Nov. 2009.

Peter M. Maurer received his Ph.D. in 1982 from Iowa State University. He has been with Baylor University since 2002. He was with the University of South Florida from 1987 through 2002, and was a member of technical staff at Bell Laboratories from 1982 through 1987. His interests are GF(2) matrices as applied to EDA problems, gate-level simulation, exotic computer architectures, and random test generation.