Symmetry Detection and Boolean Matching Utilizing a Signature-Based Canonical Form of Boolean Functions

Afshin Abdollahi, member and Massoud Pedram, Fellow

***Abstract* - A compact canonical form and a computational procedure for solving the Boolean matching problem under permutation and complementation of variables are presented. The proposed approach, which utilizes generalized signatures and variable symmetries, can handle combinational functions with no limitation on the number of input variables. Experimental results demonstrate the generality and effectiveness of the proposed canonical form and the associated Boolean matching algorithm.**

I. INTRODUCTION

BOOLEAN matching is the problem of determining whether a given Boolean function is functionally equivalent to a target function under input permutation and/or complementation of some of its input variables. Boolean matching algorithms have many applications in verification and logic synthesis. As an example, during the cell-library binding process, it is necessary to repeatedly determine whether some cluster of a Boolean network can be realized by any logic cell in a standard cell library [1].

Boolean functions that are equivalent under negation (permutation) of inputs are said to be N-equivalent (P-equivalent). Functions that are equivalent under both negation and permutation of their inputs are called NP-equivalent [2]. Equivalence under permutation and complementation of inputs as well as complementation of the output gives rise to the notion of *NPN-equivalent* Boolean functions. An exhaustive method for solving the Boolean matching problem is computationally intractable since the complexity of such an algorithm for matching two n-variable functions is $O(n!2^{n+1})$.

Boolean matching algorithms may be classified into two categories: algorithms that utilize pair-wise matching and those based on canonical forms of functions. Pair-wise Boolean matching algorithms are based on a semi-exhaustive search where the search space is pruned by using appropriate *signatures* (filters). These filters tend to capture intrinsic characteristics of a Boolean function [1] and, if at all possible, are independent of the permutation or complementation of the function variables. Canonical form-based Boolean matching algorithms work by computing some complete and unique (*canonical*) forms of the Boolean functions. The idea is that two functions match if and only if their canonical forms are the same.

The power of canonical form-based Boolean matching is best manifested in the cell-library binding application. At the first stage of the process, i.e., the library preprocessing step, canonical forms of the library cell functions are computed. For efficient equivalence checking of canonical forms, a hash table is utilized to store the canonical forms of all library cell functions. This preprocessing is performed only once for a given library. During the cell binding step, to find a cell that covers a subgraph of the subject graph, the canonical form of the cluster function is computed. Next, the hash table is checked for the presence of the canonical form of the cluster function. A matching is found if and only if the canonical form of the cluster function is in the table. This method thereby eliminates the need for pair wise matching of the cluster against the library cells one cell at a time.

Burch and Long introduced a canonical form for matching under input complementation and a semi-canonical form for matching under input permutation [3]. In their solution, to simultaneously handle complementation and permutation of inputs, a large number of forms for each cell are required. Other researchers, including Wu et al. [4], Debnath and Sasao [5], and Ciric and Sechen [6] have proposed canonical forms that are applicable to Boolean matching under permutation of the variables only. Hinsberger and Kolla [7] and Debnath and Sasao [8] have introduced a canonical form for solving the general Boolean matching problem. However, their approach is mainly based on manipulation of the truth table of the function and by employing a table look-up, which results in an enormous space complexity, thus restricting their algorithm to library cells with seven or fewer input variables. Mohnke and Malik [9] introduced an approach which computes a signature for each variable or phase of a variable, which is subsequently helpful in establishing the correspondence of variables or phases of variables. However, according to their reported results, their approach fails to conclude a unique correspondence of variables or phases of variables for some of the benchmark circuits. Chai and Kuehlmann [10] presented a matcher by integrating a number of different techniques from previous works and adding new heuristics. The authors of [10] however, do not provide results for circuits with larger number of inputs due to the space complexity of their method.

The present paper introduces a new canonical form for representing Boolean functions. The proposed canonical form of a Boolean function is the unique Boolean function that is obtained after applying some canonicity-producing (CP) transformation on its input and output variables. The proposed transformation is based on utilizing generalized signatures (signatures of one or more input variables) to find a phase assignment and total ordering for the input variables. In this paper we extend our preliminary work in [21]. Some of the key differences and extensions are as

follows. In [21] we only handled input phase assignments and permutation while in this extended version, we handle output phase assignments as well. We have significantly changed the presentation and notation throughout the paper and provided more details about the proofs of lemmas and theorems. We have added an entire section on symmetries and on the importance of symmetries in the Boolean matching problem. Finally we have explained how to detect symmetries by our proposed algorithm.

In the remainder of this paper, a combination of phase assignment and ordering for input variables is referred to as a *transformation* on variables. For most Boolean functions, single- and two-variable signatures are sufficient to *recognize* all variables (i.e., produce a CP transformation.) However, use of single-variable and two-variable signatures alone may not in general result in a canonical input transformation.

The canonical form is defined based on a property that makes it unique among all functions in an NPN-equivalence class. The key task is to devise a canonical form that handles permutation and complementation of inputs and output with a low (average) time complexity. This is achieved by using the concept of a signature vector with two important properties: (i) the signature vector of a function is unique, (ii) the signature vector may be used to define a total ordering on Boolean functions. We will show that the canonical form (NPN-representative) of an NPN-equivalence class is the function that is the greatest in the class according to this ordering.

A number of pervious researchers have used the notion of signatures to address the Boolean matching problem. For example in [11] the authors have introduced the notion of a "universal" signature, which is defined in terms of a single variable of a Boolean function. Unfortunately, for many functions, such signatures fail to generate a canonical form. As an example, consider $f(x_1,x_2,x_3,x_4) = x_1x_2+x_2x_3+x_3x_4+x_4x_1$, the universal signatures of x_1, x_2, x_3, x_4 are identical; hence it is impossible to derive a canonical form for this function by using its universal signatures. In contrast, in this paper we introduce a signature vector which is defined for a function with respect to all groupings of its variables.

Other researchers [12][13] have used the Walsh spectrum for defining the canonical form. These methods require the computation of the entire Walsh spectrum (which is an integer vector of size 2^n for an n-input function) and processing this vector. In contrast, in the proposed method, often only a small portion of the complete signature vector is computed. As stated earlier, the 0^{th}, 1^{st}, and 2^{nd} order signatures are sufficient in most cases. Another advantage of our proposed method is its efficient handling and employment of symmetry relations. More precisely, our method takes advantage of variable symmetry and signatures of variables (mostly 1^{st} and 2^{nd} order signatures) to efficiently (on average only) compute the canonical form of a Boolean function under the NPN equivalence relation. This efficient computation is made possible because of a number of important properties of the proposed canonical form. In fact the proposed canonical form is defined so that it possesses properties, which can be exploited for efficient computation of the canonical form.

A. *Overview of the proposed algorithm*

The process of computing the canonical form, F, for a given function, f, is regarded as applying a transformation – i.e., a complementation of some inputs of f (input phase assignment), a permutation on its inputs (input permutation), and a complementation of output of f (output phase assignment) – that converts f to its canonical form F.

Before we define the signatures of a Boolean function, we can describe some important properties of the canonical form, F, with respect to its 0^{th} and 1^{st} signatures. These properties are as follows. (i) The 0^{th} signature of the canonical form, F, of any given function, f, is greater than or equal to that of the complement of F. This property is used for computing the output phase assignment; (ii) The 1^{st} signature of the canonical form, F, with respect to any of its variables, is greater than or equal to that of the complement of that variable. This property is used for computing the input phase assignment; (iii) The 1^{st} signatures of the canonical form, F, with respect to input variables are sorted non-decreasingly (the input variables of a function are assumed to be indexed.) This property is used for computing the proper permutation on inputs; and (iv) Symmetric variables of the canonical form, F, appear consecutively in the inputs of function F. Since swapping symmetric variables does not change the functionality, the search space for input permutations is significantly reduced due to this property.

If these steps do not result in a unique transformation (i.e., there is a tie among the 1^{st} signatures), the 2^{nd} and (when necessary) higher order signatures will be used to break the ties. There may be more than one transformation that converts a function to its canonical form. This situation occurs because of the existence of symmetries.

The proposed algorithm for computing the canonical form returns all transformations that convert a function into its canonical form. One can use the relationships among these canonicity-producing transformations to construct all possible functional symmetry relations for the given function. A key advantage of the proposed technique is the way it handles and uses the symmetry of variables to minimize the complexity of the Boolean matching algorithm compared to some of the previous approaches, which are not able to consider symmetries [7][8].

In section II, first terminology and some key definitions are provided followed by a description of symmetry relations and signatures used in the paper. In section III, the canonical form is defined and the algorithm for computing the canonical form is described. Experimental results and conclusions are given in sections IV and V.

II. PRELIMINARIES

Let $X = (x_1, x_2,\ldots, x_n)$ denote a vector of Boolean variables and $f(X)$ a single-output completely-specified Boolean function of X. A literal is a variable, x, or its complement $\bar{x}$. We denote literals by simple letters such as y, which does not necessarily mean that the phase of literal is positive. A cube is the Boolean conjunction of literals. A

minterm is a cube with n literals. | f | denotes the number of minterms covered by f.

Definition 1: An NP transformation T on a vector X is defined as a phase assignment followed by a permutation. The inverse of T, is a transformation T^{-1} such that applying T and T^{-1} successively to X results in X.

Γ_n denotes the set of NP transformations on a vector of size n.

An example of an NP transformation on (x_1, x_2, x_3) is $(\bar{x}_2, \bar{x}_3, x_1)$. In the remainder of this paper we denote a transformation T and the result of applying T to vector $X = (x_1, x_2, \ldots, x_n)$ by TX.

The cofactor of f with respect to a literal y, denoted by f_y, is the function obtained by setting y to 1 in f. The cofactor of f with respect to a cube c, denoted by f_c, is the function obtained by setting all literals of the cube to 1.

Definition 2: Two functions $f(X)$ and $g(X)$ are NPN-equivalent ($f \equiv g$), if there exists an NP transformation T so that $f(X)$ equals $g(TX)$ or its complement.

Example 1: Let $f = x_1 + x_2 x_3$ and $g = x_3(x_1 + \bar{x}_2)$. It is easy to see that $f(X) = \overline{g(TX)}$ (or $\bar{g}(TX)$) where $T = (\bar{x}_3, x_2, \bar{x}_1)$. Thus, $f(X)$ and $g(X)$ are NPN-equivalent.

NPN-equivalence is an equivalence relation, which partitions the set of all single output Boolean functions into equivalence classes. Boolean matching is often defined in terms of P, NP or NPN-equivalence. In principle, NPN-equivalence can be reduced to $2^{n+1}n!$ tautology checks.

A. Symmetry relations

Functional symmetries provide significant benefits for multiple tasks in synthesis and verification [14]-[20]. As will be explained below, concepts of Boolean matching and symmetry are closely related. In the proposed Boolean matching algorithm, this relationship manifests itself in two ways. First, simple types of symmetries (that are inexpensive to discover) are utilized to reduce the complexity of the Boolean matching algorithm. Second, the proposed Boolean matching algorithm will generate (as a bi-product) the remaining (more complicated) symmetries.

Symmetries provide insights into the structure of the Boolean function, which can subsequently be used to facilitate operations on it. Symmetries may also serve as a guide for preserving that structure when the function is transformed in some way. In the context of the Boolean matching problem, symmetries that we explore are variable permutations, with possible complementations, that leave the function unchanged or simply invert the function. In the presence of functional symmetries, several design problems (e.g., circuit restructuring, checking satisfiability, and computing sequential reachability) are considerably simplified. Hence, interest in functional symmetries started in the early days of logic design [14] and has continued until now [15]-[20]. In [16], functional symmetry is exploited to optimize a circuit implementation for low power consumption and delay under an area increase constraint. Another benefit of knowledge about functional symmetries is that it can help produce better variable orders for Binary Decision Diagrams (BDDs) and related data structures (e.g., Algebraic Decision Diagrams). The size of the BDD of a Boolean function can be significantly reduced if symmetric variables are placed in adjacent positions [17]. This plays a crucial role in BDD-based symbolic model checking.

In the physical design domain, functional symmetries are used to improve rewiring, re-buffering, and post-placement optimization [18] [19]. The authors of [22] and [23] utilize automorphisms for symmetry identification.

We consider symmetries in the most general from, i.e., considering input permutation, input phase assignment, and output phase assignment which has not been studied thoroughly enough by other researchers in the past.

Definition 3: A function f is symmetric with respect to an NP transformation T if $f(X)$ equals $f(TX)$ or $\bar{f}(TX)$. We refer to such a transformation (T) as a *symmetry-producing* (SP) transformation.

We denote the set of all SP transformations by S_f, which creates a sub-group of Γ_n. As mentioned earlier, some types of symmetry are easily detectable and are discovered before the Boolean matching algorithm. We start by discussing *simple* symmetries.

Definition 4 (*Simple Symmetry*): Two literals x and y are said to be symmetric in f, denoted as $x \equiv y$, if f is invariant under an exchange of x and y.

Example 2: Given $f(X) = (x_1 + x_2)(\bar{x}_3 + x_4)$, we have $x_1 \equiv x_2$ and $x_3 \equiv \bar{x}_4$.

It is well known, and can be readily shown by using the Boole's expansion theorem [5], that condition $x \equiv y$ is equivalent to $f_{x\bar{y}} \equiv f_{\bar{x}y}$. The variable symmetry relation is an equivalence relation. Hence, it is possible to partition variables $x_1, x_2, \ldots, x_n$ into equivalence classes, which we will refer to as *symmetry classes* $C_1, C_2, \ldots, C_m$. The phases of variables in classes are chosen so that if two literals x and y belong to the same class, then $x \equiv y$.

Example 3: For function $f(X) = (x_1 + x_2)(\bar{x}_3 + x_4)$, there exist two symmetry classes: $C_1 = \{x_1, x_2\}$ and $C_2 = \{x_3, \bar{x}_4\}$.

There are a number of algorithms in the literature for generating symmetry classes e.g., [20].

So far we have discussed simple symmetries which correspond to NP transformations that involve only two variables. In the sequel we present a key theorem, which provides a valuable insight for handling and enumerating symmetries. We investigate the effect of the SP transformation T on simple symmetry classes.

Theorem 1: Let function f be symmetric with respect to SP transformation T and C_k be a symmetry class of variables of f. Then, mapping literals of C_k under T gives rise to a symmetry class.

Proof: The composition of SP transformations is an SP transformation. Lets T_1 denote swapping of two literals x and y. Let x' and y' denote mappings of x and y under T. The reader can verify that TT_1T^{-1} denotes swapping of x'

and y' and since TT_1T^{-1} is an SP transformation x' and y' are symmetric. The theorem follows from this fact.■

The theorem states that any SP transformation maps symmetry classes to other symmetry classes. This result, which may be considered as a constraint for any SP transformation, is especially important in the process of identifying SP transformations because it limits the space of transformations that must be explored. More precisely, to explore possible SP transformations, it is sufficient to explore only NP transformations that are specified in terms of higher order symmetry classes instead of the individual variables. Since the class count is usually considerably smaller than the variable count, this theorem tends to greatly reduce the search space.

B. Signatures

Conventionally, a signature represents a (quantitative) characteristic of a Boolean function with respect to one or more of its variables. For example, the onset size signature provides the number of minterms in the onset of a Boolean function. In the context of Boolean matching, signatures are frequently used as necessary conditions for the matching of two logic functions. For example, two functions that do not have the same onset size signatures are clearly different functions. However, even when they have the same signatures, they can be different. A signature that depends on only one input variable is called a first order signature (or 1st-signature). The 1st-signatures have been traditionally defined with respect to variables. However, since we intend to consider phase assignment in addition to permutation of variables, we define the 1st-signatures with respect to literals.

A well-known 1st-signature for a literal x of a Boolean function f is the "minterm" count of the ONSET of the cofactor of this function with respect to x i.e., $|f_x|$. In pair-wise matching methods (for checking P-equivalence), a 1st-signature must be able to recognize an input variable x_i independent of any input variable permutation so that it can establish a correspondence between variable x_i of f with a variable y_j of some other Boolean function g. It makes sense to try to establish a correspondence between these two variables only if they have the same 1st-signatures.

The main idea of the pair-wise matching approach is now evident: if we are able to compute a unique signature for each input variable of f, then the variable mapping problem will be solved because there will be either exactly one or possibly no variable correspondence for the P-equivalence of function f with respect to some other function g. More precisely, if, for each variable of f, we find a variable of g that has the same unique signature, then we will establish a one-to-one correspondence between variables of f and g. Otherwise, we will know that these two functions are not P-equivalent. The main difficulty that arises in this paradigm is when more than one variables of function f have the same 1st-signature. In such a case, it is not possible to distinguish between these variables, i.e., there is no unique correspondence that can be established with the inputs of some other function. We will thus generalize the concept of 1st signatures to higher order signatures and define a signature vector that has the full expressive power to handle the Boolean matching problem. The expressive power of the signature vector is not the only motivation for this approach. Another incentive is that the canonical form defined by using the proposed signature vector possesses a number of properties, which significantly reduce the computational complexity of obtaining the canonical form.

Definition 5: The k^{th} order signature of function f with respect to literals l_1, l_2, …, l_k is the minterm count of cofactor of f with respect to cube c= $l_1l_2\ldots l_k$, i.e., $|f_c|$. The 0^{th} order signature is $|f|$.

Definition 6: For a function f with n variables in its input support set, the signature vector denoted by V^f includes the 0^{th}-signature followed by the 1st-, 2nd- and higher order signatures up to the n^{th}-signature.

$$V^f = \Big(|f|, \overbrace{|f_{x_1}|, |f_{x_2}|, \ldots, |f_{x_n}|}^{1^{st}-\text{signatures}}, \overbrace{|f_{x_1x_2}|, |f_{x_1x_3}|, \ldots, |f_{x_{n-1}x_n}|}^{2^{nd}-\text{signatures}}, \ldots,$$
$$\overbrace{|f_{x_1\ldots x_{n-1}}|, \ldots, |f_{x_2\ldots x_n}|}^{(n-1)^{st}-\text{signatures}}, \overbrace{|f_{x_1\ldots x_n}|}^{n^{th}-\text{signature}}\Big)$$

Next we present an important theorem, which proves that the signature vector of a function is unique i.e., different Boolean functions have different signature vectors.

Theorem 2: For a function f, signature vector V^f uniquely and completely specifies function f.

Proof: The value of function V^f for all minterms can be obtained from signatures in V^f. Some of the computations are as follows:

$$f(1,1,\ldots,1) = |f_{x_1x_2\ldots x_n}|$$
$$f(0,1,\ldots,1) = |f_{\bar{x}_1x_2\ldots x_n}| = |f_{x_2\ldots x_n}| - |f_{x_1x_2\ldots x_n}|$$
$$|f_{\bar{x}_2x_3\ldots x_n}| = |f_{x_3\ldots x_n}| - |f_{x_2x_3\ldots x_n}|$$
$$f(0,0,1,\ldots,1) = |f_{\bar{x}_1\bar{x}_2\ldots x_n}| = |f_{\bar{x}_2x_3\ldots x_n}| - |f_{x_1\bar{x}_2x_3\ldots x_n}|$$
$$f(0,0,1,\ldots,1) = |f_{x_3\ldots x_n}| - |f_{x_2x_3\ldots x_n}| - |f_{x_1x_3\ldots x_n}| + |f_{x_1x_2x_3\ldots x_n}|$$

Other minterms are similarly obtained. ■

III. THE SIGNATURE-BASED CANONICAL FORM

Let's consider an NPN-equivalence class, $EC=\{f_1,f_2,\ldots,f_m\}$, of n-input Boolean function. Any two functions in EC are NPN-equivalent and any function that is NPN-equivalent to some function in EC is also in EC. The Boolean matching problem under NPN-equivalence is reduced to that of verifying whether two target Boolean functions, f and g, belong to the same NPN-equivalence class.

Let's denote the canonical form of a function f by F (capital letters are used for canonical forms e.g., the canonical form of a function f_i is denoted by F_i.) In the canonical form based Boolean matching, a unique representative, called the NPN-representative of the class, is selected for every class as formalized in the next definition.

Definition 7: The canonical form of functions f_1, f_2,…, f_m in an NPN-equivalence class EC is defined as the NPN-representative, F, of EC. Clearly, $F \equiv F_1 = F_2 = \ldots = F_m$.

The NPN-representative F is selected based on some criteria that make F unique in EC. One way is to define a total ordering for functions in EC and select the maximum or the minimum (with respect to the defined order) as the

NPN-representative (canonical form) of the class.

Fact: Two functions f and g are NPN-equivalent if and only if they have the same canonical form.

The NPN-equivalence class that includes a function f, denoted by E_f, is the set of all functions that are NPN-equivalent to f. Hence, E_f may be created by applying the set of all NP transformations and output phase assignments to f one at a time.

Definition 8: Given function f, an NP transformation T such that $F(X)$ is equal to $f(TX)$ or its complement is called a canonicity-producing (CP) transformation.

We present an algorithm to compute the canonical form of a given NPN-equivalence class as well as the set of all CP transformations C_f. We will show that the set of symmetry-producing (SP) transformations S_f can be easily obtained from C_f. The importance of identifying all NP transformations in S_f was explained earlier. For a set S of NP transformations and a given transformation T, we define ST, i.e., the *right coset* of S determined by T, as the composition of all transformations in S with T.

Lemma 1: For a function f, let T and T' be two CP transformations. Then $T'T^{-1}$ and is an SP transformation.

Proof: Clearly, $F(X)$ is equal to $f(TX)$ or its complement and also to $f(T'X)$ or its complement, which means that $f(X)$ is equal to $f(T'T^{-1}X)$ or its complement. ■

Theorem 3: For a function f and any CP transformation T, C_f is the right coset of S_f determined by T.

Proof: It follows directly from Lemma 1. ■

The set of SP transformations, S_f, includes transformations that correspond to simple symmetries. In the algorithm that we will present next to identify CP transformations, C_f, simple symmetries are first identified because their computational complexity is lower than that of computing the general symmetries. This information is subsequently used to compute C_f. Finally, based on C_f, the remaining SP transformations of S_f are computed.

A. Proposed canonical form and its properties

As mentioned earlier, among functions of an NPN-equivalence class, the NPN-representative is selected based on a criterion that makes the representative unique among all functions in the class. We defined the signature vector for a function and proved that it is unique for every function. We define a total ordering for functions based on a lexicographical comparison of their signature vectors.

Definition 9: Let '$\leq$' denote the lexicographic comparison of vectors. Consider two functions f and g with signature vectors V^f and V^g, respectively. The order relation '$\leq$' between f and g is defined as: $f \leq g$ if and only if $V^f \leq V^g$.

Using this order relation, the NPN-representative (canonical form) is defined as follows.

Definition 10 (NPN-representative) Consider an NPN-equivalent class of functions $EC = \{f_1, f_2, \ldots, f_m\}$ defined on variable set $y_1, y_2, \ldots, y_n$. Let S be a subset of EC where every function $f_i(y_1, y_2, \ldots, y_n)$ in S satisfies the following condition:

(i) if $y_i \equiv y_j$ for some $i < j$, then $y_i \equiv y_{i+1} \equiv \ldots \equiv y_j$.

We define the representative of EC (called the NPN-representative) as a function F in S such that:

(ii) for all functions f_i in S, $f_i \leq F$.

Condition (i) ensures that for the canonical form F of a class EC, symmetric variables are positioned consecutively in $(y_1, y_2, \ldots, y_n)$, i.e., variables will be arranged as:

$$\overbrace{y_1, y_2, \cdots, y_{n_1}}^{C_1}, \overbrace{y_{n_1+1}, y_{n_1+2}, \cdots, y_{n_1+n_2}}^{C_2}, \cdots, \overbrace{y_{n-n_k+1}, y_{n-n_k+2}, \cdots, y_n}^{C_k}$$

where $C_1, C_2, \ldots, C_k$ are symmetry classes and $n_i = |C_i|$. Condition (ii) guarantees that the canonical form F is maximal according to the relation '$\leq$' among all functions that satisfy (i).

In this section we detail some important properties of the proposed canonical form, which may be used to compute the canonical form. We will use the vector $Y = (y_1, y_2, \ldots, y_n)$ to represent the inputs of the canonical form F. This notation helps the reader better understand the process.

Theorem 4: Let $F(Y)$ be the canonical form of an NPN-equivalence class EC and F_{y_i} denote the cofactor of F with respect to y_i. We have:

(i) $F \geq \overline{F}$ (Corollary: $|F| \geq |\overline{F}|$.)

(ii) $F_{y_i} \geq F_{\overline{y}_i}$ (Corollary: $|F_{y_i}| \geq |F_{\overline{y}_i}|$.)

(iii) If $y_i \equiv y_j$ then $|F_{y_i}| = |F_{y_j}|$; Otherwise,

for $i<j$, $|F_{y_i}| \geq |F_{y_j}|$ and if $|F_{y_i}| = |F_{y_j}|$ then

$$(|F_{y_i y_1}|, \ldots, |F_{y_i y_{i-1}}|, |F_{y_i y_{i+1}}|, \ldots, |F_{y_i y_{j-1}}|, |F_{y_i y_{j+1}}|, \ldots, |F_{y_i y_n}|) \geq$$
$$(|F_{y_j y_1}|, \ldots, |F_{y_j y_{i-1}}|, |F_{y_j y_{i+1}}|, \ldots, |F_{y_j y_{j-1}}|, |F_{y_j y_{j+1}}|, \ldots, |F_{y_j y_n}|)$$

Proof: (i) It follows from the definition of the canonical form (condition (ii) of definition 10.) (ii) The proof is by contradiction. Assuming that $F_{y_i} \geq F_{\overline{y}_i}$ is not correct for some i, then negating y_i and other variables in the symmetry class of y_i will transform F to another function F' with a greater signature vector than F resulting in $F' \leq F$ to be false, which is a contradiction. (iii) The proof is by contradiction. Assuming that assertion (iii) is false, swapping y_i with y_j (along with other variables in symmetry classes of y_i with y_j to meet condition (i) of definition 10) will transform F to another function F' that satisfies condition (i) of definition 10 and $F' > F$ i.e., contradicts condition (ii) of definition 10. ■

Part (iii) implies that the 1st-signatures are sorted non-increasingly, i.e.:

$$|F_{y_1}| = \cdots = |F_{y_{n_1}}| > |F_{y_{n_1+1}}| = \cdots = |F_{y_{n_1+n_2}}| > \cdots > |F_{y_{n-n_k+1}}| = \cdots = |F_{y_n}|$$

B. Computing the canonical form

Given a function f, the goal is to find its canonical form F and the corresponding set of CP transformations, C_f. Theorem 4 imposes conditions on the canonical form F. Our approach is to project conditions on F into conditions on CP transformation, T. This greatly reduces the search space.

The proposed algorithm, called *compute_*C_f, uses signatures of function f to compute the CP transformations on inputs and the corresponding output phase assignments. In most cases, the 0th and 1st-signatures determine the inequalities required to identify the desired NP transformation. If unsuccessful, the remaining

comparisons are performed by using the 2nd-signatures and/or higher order signatures.

Experimental results indicate that in the great majority of cases, a signature inequality occurs for the low order signatures (0th, 1st and 2nd signatures.) Intuitively, the reason is that the lower order signatures depend on a higher number of minterms of the function, and thus, contain more information about the function, e.g., a 1st-signature depends on 2^{n-1} minterms, which is half of the whole Boolean space with 2^n minterms, whereas a 2nd-signature depends on 2^{n-2} minterms. Hence, the 1st-signatures are the most powerful signatures. The 2nd-signatures are the next most useful signatures, and so on. This arrangement of the proposed signature vector minimizes the computational complexity.

The first step of the *compute_*C_f algorithm is to identify the output phase assignment. If $|f| \neq |\bar{f}|$, the output phase can be uniquely determined (i.e., the phase that results in the larger value is chosen.) However, if $|f| = |\bar{f}|$, the output phase is undecided, and will be determined in subsequent steps of the algorithm. For now, we assume that the output phase is decided (the other case will be discussed afterwards) and, for simplicity, use f to denote the output function after phase assignment.

Algorithm *compute_*C_f ($f(X)$)
Input: A Boolean function $f(X)$
Output: The canonical form $f(X)$ and CP transformations C_f.
using the 0th-signature perform output phase assignment;
using the 1st-signatures
  if $|f_{x_i}| > |f_{\bar{x}_i}|$ assign positive phase to x_i;
  else if $|f_{x_i}| < |f_{\bar{x}_i}|$ assign negative phase to x_i;
  else mark the phase of x_i as undecided.
create symmetry classes and order the classes such that: $|f_{C_1}| \geq |f_{C_2}| \geq \cdots \geq |f_{C_m}|$
group classes to groups $G_1,G_2,...,G_k$ such that all classes inside a group have the same 1st-signature:
recursive_resolve($G_1,G_2,...,G_k;C_f$) ;
set the canonical form: $F(X)=f(TX)$ where $T \in C_f$;

In the next step, input phase assignment is performed by using the 1st-signatures. For variable x_i, if $|f_{x_i}| > |f_{\bar{x}_i}|$ (or $|f_{x_i}| < |f_{\bar{x}_i}|$), then a positive (or negative) phase is selected for x_i. However, if $|f_{x_i}| = |f_{\bar{x}_i}|$, then the phase of x_i remains undecided. Undecided input phases will be determined in subsequent steps of the algorithm. Let's rename the variables such that y_i denotes variables after phase assignment. Note that if the phase of some variable x_i is undecided, we define $y_i=x_i$ but record that the input phase is undecided. We sort signatures $|f_{y_i}|$ in a non-increasing order and re-index y_i's so that: $|f_{y_1}| \geq |f_{y_2}| \geq \cdots \geq |f_{y_n}|$.

In the next step, symmetry classes of variables are determined. A necessary condition for two variables y_i and y_j to be symmetric is that they have the same 1st-signatures. If phases of variables y_i and y_j are undecided, we will determine y_i and y_j to be symmetric only if they are symmetric independent of their phases. An example of this situation occurs when f depends on $y_i \oplus y_j$, i.e., $f = g((y_i \oplus y_j), y_1,\ldots,y_{i-1},y_{i+1},\ldots,y_{j-1},y_{j+1},\ldots,y_n)$. Based on these symmetry relations, we form the symmetry classes of variables $C_1,C_2,\ldots,C_m$. Function f will remain invariant under permutations inside a symmetry class. Based on this fact and since symmetric variables are positioned consecutively in the inputs of function F, instead of finding NP transformations on variables, it is sufficient to search for NP transformations on classes $C_1,C_2,\ldots,C_m$, which greatly reduces the size of search space. This method returns CP transformations *modulo* simple symmetries, i.e., when a number of CP transformations are related to each other by a simple symmetry relation (variable swapping), then any one of these transformations will be returned. Next we discuss the concept of NP transformations on classes.

Phases of classes that contain variables with decided phases are positive. The phase assignment for classes that contain variables with undecided phases can be toggled by toggling the phases of all variables in the class. An NP transformation on classes $C_1,C_2,\ldots,C_m$ signifies a transformation on variables $x_1,x_2,\ldots,x_n$. The cofactor of function f with respect to any member of class C_i is a unique function; hence, the cofactor of f with respect to class C_i can be defined as f_x for any $x \in C_i$. Similarly, the 1st-signature of f with respect to class C_i may be defined as $|f_x|$. In the next step, classes are ordered based on their 1st-signatures. Let's re-index the classes so that: $|f_{C_1}| \geq |f_{C_2}| \geq \cdots \geq |f_{C_m}|$. If these 1st-signatures are distinct, then a unique ordering will be achieved, in which case the algorithm terminates, returning a CP transformation that results from reordering and phase assignment. Otherwise, the classes are placed in *k groups* such that all classes inside a group have the same 1st-signature:

$$\overbrace{C_1,\cdots,C_{n_1}}^{G_1},\overbrace{C_{n_1+1},\cdots,C_{n_1+n_2}}^{G_2},\cdots,\overbrace{C_{n-n_k+1},\cdots,C_m}^{G_k}.$$

We refer to a group as *unresolved* if the group contains more than one class or phases of classes in the group are undecided. If all groups are resolved, a unique ordering will be obtained and the algorithm will terminate. The goal thus is to resolve all unresolved groups. Let $G_j=\{C_j,C_{j+1},\ldots,C_l\}$ be the first unresolved group. Since all groups $G_1,G_2,\ldots,G_{j-1}$ have been resolved, (i.e., they contain a single class with a decided phase), the ordering of classes up to G_{j-1} is known. (The case when G_1 is unresolved is discussed at the end of next paragraph.) Now the 2nd-signatures are used to specify the ordering inside the unresolved groups starting with G_j. Since G_1 is resolved, $G_1=\{C_1\}$, the 2nd-signatures with respect to C_1 and C_i for $j \leq i \leq l$ can be used for phase assignment (if needed) and ordering of classes $C_j,C_{j+1},\ldots,C_l$ (this step is referred to as iteration 1.)

If phases of classes C_i in G_j are undecided, the 2nd-

signatures $|f_{C_lC_i}|$ and $|f_{C_l\bar{C}_i}|$ are compared to decide the phase for C_i. In case of equality of the 2nd-signatures, the phase of C_i remains undecided. Next, new values of 2nd-signatures $|f_{C_lC_i}|$ after phase assignment are used to order classes $C_j,C_{j+1},\ldots,C_l$ and later regroup these classes. Subsequently, G_j is split into smaller groups such that inside each group the 2nd-signatures are equal. The same procedure (phase assignment, ordering and regrouping based on 2nd-signatures) is applied to all other unresolved groups. Finally, the indices of new groups are properly updated. If after these steps, there still exists some unresolved group, G_l, a similar procedure will be applied based on the 2nd-signatures with respect to C_2 and $C_i \in G_l$ (this is named iteration 2.) If needed, iterations $3,4,\ldots,j-1$ will be performed. If, at iteration j, there still exists some unresolved groups and G_j itself is also unresolved, the procedure described below will be used. This case includes the case where G_1 is unresolved.

At this point, groups $G_1,G_2,\ldots,G_{j-1}$ are resolved. However, group $G_j=\{C_j,C_{j+1},\ldots,C_l\}$ are not resolved (since the 1st and 2nd signature have not made them distinct.) There are $l-j+1$ ways (for $j\le i\le l$) to split G_j into two groups: new $G_j=\{C_i\}$ and new $G_{j+1}=\{C_j,\ldots,C_{i-1},C_{i+1},\ldots,C_l\}$. If the phase of C_i is undecided, then there will be two ways to resolve the new group, G_j. Consequently, there are $r=l-j+1$ ways (or in the worst case $r=2(l-j+1)$ ways) to specify and resolve the new group, G_j. All these r cases need to be tracked, since it is unknown which one(s) will result in a CP transformation. For each case, the 2nd-signatures, $|f_{C_jC_i}|$, are used to first order classes inside the unresolved groups among $G_{j+1},G_{j+2},\ldots,G_m$ and then split them based on the outcome of ordering. This process continues for all r cases recursively (cf. the *recursive_resolve* algorithm) until all groups are resolved. Each case corresponds to a different input phase assignment and re-indexing. Any input phase assignment and re-indexing transforms $X=(x_1,x_2,\ldots,x_n)$ to $Y=(y_1,y_2,\ldots,y_n)$ where $F(Y)=f(X)$. Each such relation can be described by an NP transformation T where $X=TY$. The r cases result in transformations $T_1,T_2,\ldots,T_s$ where in general $s\ge r$ because each case returns more than one transformation as a result of the recursion process.

Because of the way the CP transformations are constructed, they will be among $\{T_1,T_2,\ldots,T_s\}$ (modulo simple symmetries). Hence, by using the 1st and 2nd signatures, we have limited the search for a CP transformation among all $2^n n!$ transformations of Γ_n to that of searching among $\{T_1,T_2,\ldots,T_s\}$, which is a significantly smaller space than Γ_n.

CP transformations among $\{T_1,T_2,\ldots,T_s\}$ are identified based on the fact that T_i is a CP transformation if and only if $f(T_iX) \ge f(T_jX)$ for $1\le j\le s$. This task requires repeated comparison between $f(T_iX)$ and $f(T_jX)$. The comparison is done based on the signature vectors. However, before using the signature vectors, the possibility of equivalency of T_i and T_j should be considered, i.e., first the relation $f(T_iX)=f(T_jX)$ should be checked since in case of equality their signature vectors will also be equal. Because of the manner by which the NP transformations $T_1,T_2,\ldots,T_s$ are obtained, they all have the same set of 0th and 1st signatures. In fact, some of their 2nd signatures are also equal. Hence, to avoid redundancy in comparing $f(T_iX)$ and $f(T_jX)$, only signatures that have not already been determined to be equal are generated and compared. Since comparison is done based on lexicographic comparison of signature vectors, signatures are generated one by one based on their effectiveness. Only in case of equality, subsequent signatures are generated and compared. Experience shows that for nearly all functions, the 1st and 2nd signatures conclude the comparison of $f(T_iX) < f(T_jX)$.

Algorithm *recursive_resolve* ($G_1,G_2,\ldots,G_k;C_f$)
Input: Ordered groups ($G_1,G_2,\ldots,G_k$)
Output: The CP transformations C_f
$i=1$; $C_f=\{\}$;
while ($i\le m$) { // m is the number of classes
  if (G_i is resolved) { // $G_i=\{C_i\}$
    for (all unresolved groups G_j){
      use signatures $|f_{C_lC_i}|$ ($C_l\in G_j$) to assign phase,
        order and split G_j;
      update indices of groups and classes;
    }
    $i=i+1$;
  } else { // $G_i=\{C_i,C_{i+1},\ldots,C_l\}$ is not resolved
    // for space limitation assume the phase of
    // $G_i=\{C_i,C_{i+1},\ldots,C_l\}$is decided
    for ($j=i$; $j\le l$; j++) {
      split G_i to groups $\{C_j\}$ and $\{C_i,\ldots,C_{j-1},C_{j+1},\ldots,C_l\}$;
      update indices of groups and classes
        ($G_1,G_2,\ldots,G_k,G_{k+1}$);
      recursive_resolve ($G_1,G_2,\ldots,G_k,G_{k+1};C_f^{TEMP}$) ;
      if ($C_f=\{\}$ or $f(T^{TEMP}X)>f(TX)$){
        //$T\in C_f$, $T^{TEMP}\in C_f^{TEMP}$
        $C_f=C_f^{TEMP}$;
      } else if ($f(T^{TEMP}X)=f(TX)$) {
        $C_f=C_f\cup C_f^{TEMP}$;
      }
    }
    return;
  }
} //At this point there are m resolved groups
T = Transformation obtained by phase assignment and ordering;
$C_f=\{T\}$;
return;

In this description of the algorithm, we did not discuss the case where the output phase may not be decided by the 0th-signature. In such a case, the following steps will have to be performed for both output phases. Let's assume that C_f is the set of NP transformations returned by the algorithm for f and C'_f is returned for $\bar{f}$. If $f(TX) > \bar{f}(T'X)$ (where $T\in C_f$ and $T'\in C'_f$), then the output phase is positive; if $f(TX) < \bar{f}(T'X)$, then the output phase is negative; and in case equality, both phases can result in the canonical form, i.e., $F(X)=f(TX)=\bar{f}(T'X)$ and the set of CP transformations is set to $C_f \cup C'_f$.

The algorithm returns C_f which in general may contain more than one NP transformations. Based on the members

of C_f, SP transformations (other than those corresponding to simple symmetries) are detected, i.e., if T and T' are CP transformations, then $T'T^{-1}$ will be an SP transformation. Equivalently, for CP transformation T, $C_f T^{-1} \subset S_f$. Other members of S_f may be generated by composing NP transformation of $C_f T^{-1}$ with simple symmetry transformations. Composition means construction of every transformation that can be generated by members of $T^{-1}C_f$ and symmetry transformations. This step may require repeated compositions.

Example 4: Consider the multiplexer function $f(X) = \bar{x}_5\bar{x}_6x_1 + \bar{x}_5x_6x_2 + x_5\bar{x}_6x_3 + x_5x_6x_4$. The 0th signature of the function and its complement are equal, i.e., $|f| = |\bar{f}| = 32$; hence, in this step the output phase cannot be determined. We must consider both phases. First we choose the positive phase. In this function, there is no symmetry between variables, i.e., each symmetry class contains only one variable. After doing input phase assignment, ordering and grouping classes (in this case, variables), two groups G_1 and G_2 are created: $G_1 = \{x_1, x_2, x_3, x_4\}$ and $G_2 = \{x_5, x_6\}$ where $|f_{x_1}| = |f_{x_2}| = |f_{x_3}| = |f_{x_4}| = 40$ and $|f_{x_5}| = |f_{x_6}| = 32$. For members in G_2, the phase is undecided since $|f_{x_5}| = |f_{\bar{x}_5}| = |f_{x_6}| = |f_{\bar{x}_6}| = 32$ whereas for G_1 the phase is decided where the 1st-signature is: $|f_{x_i}| = 40$. Both groups G_1 and G_2 are unresolved. Group G_1 can be split in four ways:

1: $\{x_1\}, \{x_2, x_3, x_4\}$ 2: $\{x_2\}, \{x_1, x_3, x_4\}$

3: $\{x_3\}, \{x_1, x_2, x_4\}$ 4: $\{x_4\}, \{x_1, x_2, x_3\}$

The algorithm keeps track of all these cases. Let's focus on one of the cases e.g., case 1 which results in the following grouping: $\{x_1\}, G_1 = \{x_2, x_3, x_4\}, G_2 = \{x_5, x_6\}$. Now we try to resolve G_1 and G_2. Since $|f_{x_1x_2}| = |f_{x_1x_3}| = |f_{x_1x_4}| = 48$, G_1 cannot be resolved at this step. As for G_2, $|f_{x_1x_5}| = 32 < |f_{x_1\bar{x}_5}| = 48$ and $|f_{x_1x_6}| = 32 < |f_{x_1\bar{x}_6}| = 48$ which implies that negative phases should be assigned to x_5 and x_6. However, $|f_{x_1\bar{x}_5}| = |f_{x_1\bar{x}_6}|$ which means that G_2 can not be resolved further, i.e., the overall grouping thus far is: $\{x_1\}, G_1 = \{x_2, x_3, x_4\}, G_2 = \{\bar{x}_5, \bar{x}_6\}$.

There are three ways to resolve G_1. Following one of these cases, for example, will result in $x_1, x_2, G_1 = \{x_3, x_4\}, \bar{x}_5, \bar{x}_6$ since $|f_{x_2x_3}| = |f_{x_2x_4}| = 48$ and $|f_{x_2\bar{x}_5}| = 48 > |f_{x_2\bar{x}_6}| = 32$ which results in $Y_{12} = (x_1, x_2, x_3, x_4, \bar{x}_5, \bar{x}_6)$ because $|f_{x_3\bar{x}_5}| = |f_{x_4\bar{x}_5}| = 32$ and $|f_{x_3\bar{x}_6}| = 48 > |f_{x_4\bar{x}_6}| = 32$. The resulting Y_{12} corresponds to a transformations T_{12} where $X = T_{12}Y_{12}$. Indices 1 and 2 in T_{12} indicate the decisions that have been made for splitting G_1 to obtain this transformation T_{12}. Had we chosen x_3 instead of x_2, we would have arrived at $Y_{13} = (x_1, x_3, x_2, x_4, \bar{x}_6, \bar{x}_5)$ (and T_{13} from $X = T_{13}Y_{13}$). However, choosing x_4 (instead of x_2) does not immediately resolve G_1 and a further step to decide between x_2 and x_3 must be performed, resulting in $Y_{142} = (x_1, x_4, x_2, x_3, \bar{x}_5, \bar{x}_6)$ and $Y_{143} = (x_1, x_4, x_3, x_2, \bar{x}_6, \bar{x}_5)$ and subsequently T_{142} and T_{143}. It turns out that $f(T_{142}X) = f(T_{143}X)$, which means that we must keep both T_{142} and T_{143} transformations at this step. However, $f(T_{12}X) = f(T_{13}X) \neq f(T_{142}X)$ which means that we should either keep T_{142} and T_{143} or keep T_{12} and T_{13}. To make this decision signature vectors of $f(T_{12}X)$ and $f(T_{142}X)$ need to be compared. All their 0th and 1st signatures and most (but not all) of their 2nd signatures are equal. Hence, to decide between T_{12} and T_{142} we first compare the remaining 2nd signatures of $f(T_{12}X)$ and $f(T_{142}X)$. Since $|f_{x_2\bar{x}_5}| = 48 > |f_{x_4\bar{x}_6}| = 32$ is the first inequality that breaks the tie, $f(T_{12}X) > f(T_{142}X)$ which shows that only T_{12} and T_{13} should be kept. These transformations are the result of selecting x_1 to spilt G_1 at the beginning. If we choose x_2, x_3 or x_4 we will obtain transformations T_{21}, T_{24}, T_{31}, T_{34}, T_{42} and T_{43}. It turns out that $f(T_{12}X) = f(T_{13}X) = f(T_{21}X) = f(T_{24}X) = f(T_{31}X) = f(T_{34}X) = f(T_{42}X) = f(T_{43}X)$, which means that all these 8 transformations are maximal. We obtained these transformations by assuming positive phase for the output. If we choose the negative phase and follow the same steps, we will obtain 8 other transformations. It turns out that these new transformations along with output negation convert f to the same function as previous transformations i.e., there are 16 CP transformations and the canonical form is $F(X) = x_5x_6x_1 + x_5\bar{x}_6x_2 + \bar{x}_5x_6x_3 + \bar{x}_5\bar{x}_6x_4$. These 16 CP transformations result in 16 SP transformations.

IV. Experimental Results

The technique presented above was implemented and ran on a computer system with a 1.7GHz Intel Xeon processor and 1GB of memory. To evaluate the effectiveness of the proposed algorithm, canonical forms are computed for all cells in a large cell library, containing a large number of complex cells with up to 20 inputs. We started with a standard cell library and augmented it with a large number of pseudo-randomly generated logic cells with different input counts. These pseudo-random functions were generated from different single output clusters taken from the subject graphs created for MCNC benchmark circuits.

Table 1 shows the worst-case and average run-times required for computing the canonical form in terms of the number of inputs; i.e., the second and third columns are the worst-case and average runtimes for all n-input cells. The fourth and fifth columns present results (after scaling to match CPU speeds) provided in references [8] and [10]. The run-times in this table are in microseconds and include data for cells with more than 3 inputs. As an example, the worst-case 20-input cell was a multiplexer with four select inputs for which the proposed algorithm

took 240 microseconds to compute its canonical form. Our results show a major improvement in run-time over [8]. Our runtimes are similar to those of [10] for circuits of nine or fewer inputs. Reference [10] performs better on some small benchmarks. Note however that the authors of [10] do not provide results for circuits with larger number of inputs due to the space complexity of their method. (This limitation is because of using hash table to store pre-computed results).

For nearly all of the cells in the library, the canonical forms were computed by using only the 0^{th}, 1^{st} and 2^{nd}-signatures. The number of 0^{th}, 1^{st} and 2^{nd}-signatures combined is $(1+n)(1+n/2)$. Only one of the cells required the use of a single 3^{rd}-signature. In spite of this, clearly for some Boolean functions, it may become necessary to generate all of the high order signatures. In this worst-case scenario, the number of generated signatures will be the power set of n which is exponential in n. However, the algorithm described above is complete and capable of handling functions that may require the use of higher order signatures for computing the canonical form. The function that required the use of the 3^{rd} signatures was a multiplexer where one of the select inputs was the output of an XOR function. A simplified version of such a function is $F(X) = (x_1 \oplus x_2)x_3 + (x_1 \overline{\oplus} x_2)x_4$.

One of the advantages of the proposed algorithm is identifying all symmetry relations for the given function. The runtime of the algorithm for any function has a direct relation with the number of non-simple symmetry relations of the function. More precisely, the higher the number of non-simple SP transformations, the higher the runtime.

Table 1. Runtimes for computing canonical forms and number of non-simple SP transformations.

Number of Inputs	Average run-time	Worst-case run-time	Ref [8] (scaled)	Ref [10] (Scaled)	Size of S_f
3	< 1	< 1	4.62	-	4
4	< 1	< 1	7.41	0.2	6
5	1	1	26.02	0.63	8
6	2	3	39.13	1.09	16
7	3	5	147.6	1.74	18
8	4	8	–	3.57	24
9	6	14	–	7.05	32
10	8	21	–	–	64
11	14	30	–	–	128
12	20	48	–	–	144
13	23	61	–	–	162
14	33	80	–	–	216
15	37	105	–	–	288
16	44	115	–	–	324
17	56	140	–	–	384
18	67	165	–	–	512
19	80	200	–	–	576
20	100	240	–	–	768

The sixth column of Table 1 reports the number of non-simple SP transformations for functions that correspond to the worst-case runtimes in the previous experiment. Note that the number of non-simple SP transformations is equal to the number of non-trivial CP transformations i.e., $|C_f|$. Despite major advances in the algorithms for Boolean matching (including this work) the time and space complexities of all these algorithms remain exponential in the worst case. In our approach the space complexity is determined by the size of the BDD used to represent the function (which is in the worst case exponential in the number of inputs of the Boolean function.) The worst-case time complexity is exponential in the number of inputs since we may have to generate all signatures of a Boolean function ranging from the 0^{th} to the n^{th} order. However, judging by the experimental results, since the BDD sizes remain reasonable and since we have never encountered a case that required more than a 3^{rd} order signature, the time complexity remains polynomial for the attempted benchmarks and many randomly generated circuits.

V. Conclusion

This paper addressed the general Boolean matching problem in which both permutation and complementation of inputs and output are considered. A new efficient and compact canonical form was defined and an effective algorithm for computing the proposed canonical form was presented. The compactness and average efficiency of the accompanying computational procedures enables this algorithm to be applicable to a wide range of circuits without large number of inputs. Experimental results demonstrated the efficacy of the proposed approach.

References

[1] L. Benini and G. De Micheli, "A survey of Boolean matching techniques for library binding," *ACM Trans. Design Automation of Electronic Systems*, vol. 2, no. 3, pp. 193–226, July 1997.

[2] M. A. Harrison, *Introduction to Switching and Automata Theory*, McGraw-Hill, 1965.

[3] J. R. Burch and D. E. Long, "Efficient Boolean function matching," in *Proc. Int'l Conf. on Computer-Aided Design*, pp. 408–411, Nov. 1992.

[4] Q. Wu, C. Y. R. Chen, and J. M. Acken, "Efficient Boolean matching algorithm for cell libraries," *Proc. IEEE Int'l Conf. on Computer Design*, pp. 36–39, Oct. 1994.

[5] D. Debnath and T. Sasao, "Fast Boolean matching under permutation using representative," *Proc. ASP Design Automation Conf.*, pp. 359–362, Jan. 1999.

[6] J. Ciric and C. Sechen, "Efficient canonical form for Boolean matching of complex functions in large libraries," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 22, no. 5, pp. 535–544, May 2003.

[7] U. Hinsberger and R. Kolla, "Boolean matching for large libraries," *Proc. Design Automation Conf.*, pp. 206–211, 1998.

[8] D. Debnath and T. Sasao, "Efficient computation of canonical form for Boolean matching in large libraries," *Proc. ASP Design Automation Conf.*, pp. 591–596, Jan. 2004.

[9] J. Mohnke and S. Malik, "Permutation and phase independent Boolean comparison," *Integration - the VLSI journal*, 16: pp. 109-129, 1993.

[10] D. Chai and A. Kuehlmann, "Building a Better Boolean Matcher and Symmetry Detector," *Proc. Design Automation and Test in Europe*, March 2006.

[11] P. Molitor and J. Mohnke. *Equivalence Checking of Digital Circuits*. Kluwer Academic Publishers 2004.

[12] D.M.Miller, "A Spectral Method for Boolean Function Matching", *Proc. European Design and Test Conference*, pp. 602, 1996.
[13] E.M. Clarke et al., "Spectral Transforms for Large Boolean Functions with Applications to Technology Mapping", *Proc. Design Automation Conference*, pp. 54-60, 1993.
[14] C. E. Shannon, "A symbolic analysis of relay and switching circuits," *American Institute of Electrical Engineers Trans*., 57:713-723, 1938.
[15] V. N. Kravets and K. A. Sakallah, "Constructive library-aware synthesis using symmetries," *Proc. Design Automation and Test in Europe*, pp. 208-213, March 2000.
[16] K. S. Chung and C. L. Liu, "Local transformation techniques for multi-level logic circuits utilizing circuit symmetries for power reduction," *Proc. Int'l Symp. on Low Power Electronics and Design*, pp. 215–220, August 1998.
[17] S. Panda, F. Somenzi, and B. F. Plessier, "Symmetry detection and dynamic variable ordering of decision diagrams," *Proc. Int'l Conference on Computer-Aided Design*, pp. 628–631, November 1994.
[18] C. W. Chang, C. K. Cheng, P. R. Suaris and M. Marek-Sadowska, "Fast Post-placement Optimization Using Functional Symmetries," *IEEE Trans. on Computer-Aided Design*, pp. 102-118, Jan. 2004.
[19] K. H. Chang, I. L. Markov and V. Bertacco, "Post-Placement Rewiring and Rebuffering by Exhaustive Search For Functional Symmetries," *Proc. Int'l Conference on Computer-Aided Design* (ICCAD), 2005, pp. 56-63.
[20] D. Möller, J. Mohnke, and M. Weber, "Detection of symmetry of Boolean functions represented by ROBDDs," *Proc. Int'l Conference on Computer Aided Design* 1993, pp.680-684.
[21] A. Abdollahi and M. Pedram, "A New Canonical Form for Fast Boolean Matching in Logic Synthesis and Verification," *Proc. Design Automation Conference*, 2005.
[22] B. D. McKay, "Practical graph isomorphism," *Congressus Numerantium*, vol. 30, pp. 45-87, 1981.
[23] P. T. Darga, M. H. Liffiton, K. A. Sakallah, and I. L. Markov, "Exploiting structure in symmetry detection for CNF," *Proc. Design Automation Conference*,2004,pp.530-534.