An Efficient NPN Boolean Matching Algorithm Based on Structural Signature and Shannon Expansion

Juling Zhang, Guowu Yang, William N. N. Hung, and Yan Zhang

Abstract—An efficient pairwise Boolean matching algorithm to solve the problem of matching single-output specified Boolean functions under input negation and/or input permutation and/or output negation (NPN) is proposed in this paper. We present the Structural Signature (SS) vector, which is composed of a 1st signature value, two symmetry marks, and a group mark. As a necessary condition for NPN Boolean matching, the structural signature is more effective than the traditional signature. Two Boolean functions, \( f \) and \( g \), may be equivalent when they have the same SS vector. The symmetry mark can distinguish symmetric variables and asymmetric variables and search multiple variable mappings in a single variable-mapping search operation, which reduces the search space significantly. Updating the SS vector using Shannon decomposition provides benefits in distinguishing unidentified variables, and the group mark and the phase collision check discover incorrect variable mappings quickly, which also speeds up the NPN Boolean matching process. Using the algorithm proposed in this paper, we tested both equivalent and non-equivalent Boolean functions on the MCNC benchmark circuit sets and the random circuit sets. In the experiment, our algorithm is two times faster than competitors when testing equivalent circuits and averages at least one hundred times faster when testing non-equivalent circuits. The experimental results show that our approach is highly effective in solving the NPN Boolean matching problem.

Index Terms—Boolean Matching, Structural Signature Vector, Shannon Expansion, Variable Mapping, Variable Symmetry.

I. INTRODUCTION

NPN Boolean matching is a significant problem in logic synthesis, technology mapping, cell library binding, and logic verification [1]. In logic verification, a key issue is to verify the equivalence of two circuit functions [2]. In both cell library binding and technology mapping, a Boolean network is transformed into an equivalent circuit using logic cells from a standard cell library [3], [4]. Boolean matching is a critical step in technology mapping as demonstrated in [5]. NPN Boolean matching can be applied to either incompletely or completely specified functions with either multiple or single outputs. In this paper, we studied the NPN Boolean matching problem for completely specified functions with a single output.

The NPN Boolean matching problem involves judging whether one Boolean function can be transformed into another through input negation and/or input permutation and/or output negation. If the Boolean function \( g \) can be obtained from \( f \) by one of the previous types of transformations, then \( f \) and \( g \) are NPN equivalent; that is, the logic circuit of \( g \) could be implemented by \( f \).

An \( n \)-variable Boolean function has \( 2^n \) input negations, \( n! \) input permutations, and 2 possible output negations. The complexity of the exhaustive method for NPN Boolean matching is \( O(n!2^{n+1}) \). However, a Boolean function with a large number of input variables is intractable to the exhaustive approach. In this paper, an effective pairwise NPN Boolean matching algorithm is proposed. A binary decision diagram (BDD) is used to represent a Boolean function because this representation is compact and previous studies have demonstrated its fast speed when computing signatures [3]. The algorithm proposed in this paper utilizes an SS vector and Shannon expansion to detect possible transformations of two Boolean functions. Our proposed Boolean function matching algorithm, the SS vector is required. An SS value is composed of a 1st signature value, two symmetry marks, and a group mark. When the SS values of any two variables are the same, variable mappings between the two variables may exist. The symmetry marks serve to distinguish symmetric and asymmetric variables. It allows the removal of many impossible variable mappings and finds multiple variable mappings in one variable-mapping search operation, which reduces the search space of Boolean matching. The group mark can solve the 1st signature value collision problem. Boolean decomposition updates the SS vectors. The updated SS vectors are then used to further search variable mappings. Phase collision check can find error variable mappings quickly. All these methods are used to speed up transformation detection. For two NPN equivalent Boolean functions \( f \) and \( g \), the goal of our algorithm is to find a transformation that can transform \( f \) to \( g(\bar{g}) \) as quickly as possible. In the algorithm, a tree structure is used to store the detected transformations. It handles transformations in depth-first search (DFS) order.

In the cell-library binding, a logic cell would be found in a standard cell library that can implement an objective circuit. SS vectors of the library cell functions can be computed and stored in advance. During the cell-library binding, we compare the SS vectors of circuit functions and then do Boolean matching when their SS vectors are the same. The probability of having the same SS vectors is lower and our
algorithm operates very quickly in judging non-equivalent Boolean functions. Therefore, our algorithm can be applied to the cell library binding. The same is true of our algorithm’s application to technology mapping.

The algorithm proposed in [6] and our algorithm both use BDD to represent a Boolean function and compute the cofactor and symmetry. Because the results of [6] reflect the state-of-the-art in NPN Boolean matching using BDD, we re-implemented the algorithm from [6] and compared its runtime with that of our algorithm on the same data set. We tested the algorithm on both MCNC benchmark circuits and on random circuits and report the runtimes for equivalent and non-equivalent Boolean matching. By comparing the results of our algorithm with the algorithm from [6], we conclude that our proposed algorithm runs twice as fast as the algorithm from [6] on general equivalent circuits, while for non-equivalent circuits, our algorithm is far faster than the algorithm proposed in [6]. Therefore, the proposed algorithm is highly effective. Moreover, it can scale to up to 22 variables in NPN Boolean matching.

The paper is organized as follows. Section II presents current research results concerning NPN Boolean matching. Section III introduces related preliminaries and defines the problem to be solved. The NPN Boolean matching algorithm is explained in Section IV. In Section V, we provide experimental results that verify the effectiveness of our algorithm. Finally, we summarize our work and outline directions for future research in Section VI.

II. RELATED WORKS

Scholars have explored numerous methods and algorithms to solve the Boolean matching problem. Boolean matching algorithms are traditionally classified into two classes: (1) algorithms based on canonical forms, (2) algorithms based on pairwise matching [6]. The key to the first approach is to construct a unified canonical form for equivalent Boolean functions. Two Boolean functions are equivalent when their canonical forms are identical. The second method is a semi-exhaustive search that uses some appropriate signatures to establish variable mapping and reduce the search space. Function signatures are properties of Boolean functions that are independent of input variable permutation and negation [7], and they are intrinsic characteristics of Boolean functions [8].

In Boolean matching based on canonical forms, a unique Boolean function is used as a representative for each Boolean equivalent class. References [9], [10], and [11] introduced NPN Boolean matching based on canonical forms. Their canonical representative was a Boolean function with a maximal truth table. Variable symmetry and DFS were utilized to reduce the search complexity in [9]. Using a look-up table and a tree-based breadth-first search, the authors of [11] presented an algorithm to compute an NP-representative. Citric and Sechen [12] proposed an efficient canonical form for P-equivalent Boolean matching. The canonical form of [12] involved the least cost for finding the representative under permutation of rows and columns in the truth table. Damiani and Selchenko [5] used decomposition trees to create the Boolean function representative. Agosta et al. [1] combined the spectral and canonical forms to exploit a transform-parametric approach that could match Boolean functions of up to 20-variables, but this method considered only input permutation. The authors of [13] proposed a fast Boolean function NPN classification using canonical forms in which a Boolean function was represented by a truth table; then, the method computed the canonical form using a cofactor, swapping variables and symmetry. Petkovska et al. [14] proposed a hierarchical method to resolve NPN Boolean classification that improved classification speed compared with [13]. However, these two papers were able to perform NPN classification only on functions with 6–16 inputs. Abdollahi and Pedram [15] proposed a high order signature and presented a P-equivalent Boolean matching algorithm. Agosta et al. [16] used shifted cofactor transformation to achieve more efficient P-equivalent matching. Abdollahi [6] constructed a NPN Boolean matching algorithm utilizing the signature-based canonical form and extended it to incompletely specified Boolean matching [3]. Although the experiment result in [6] was fast, only the execution time for establishing the canonical form was published. Therefore, in this study, we re-implemented the algorithm from [6] and compared it to the runtime of our NPN Boolean matching algorithm.

The key to creating a pairwise matching algorithm is to establish a one-to-one variable correspondence for two Boolean functions. In this method, the cofactor signature is universally used to search for variable correspondence. But some variable correspondences cannot be determined using only the cofactor signature. Mohlke and Malik [7] exploited an NPN Boolean matching algorithm that used a breakup signature and Boolean difference when the cofactor signature failed. The authors of [6] expanded the cofactor signature to an n-order signature. Lai et al. [17] used a level-first search to complete Boolean matching and proposed a set of filters to improve the algorithm’s performance. The consensus and smoothing operators were applied to Boolean matching with a don’t-care set by Chen [2]. Wang and Hwang [18] developed an efficient Boolean matching algorithm using cofactor and equivalence signatures based on communication complexity. The authors of [4], [19], [20], [8] presented a pairwise Boolean matching utilizing don’t-care sets in technology mapping.

There are other approaches to Boolean matching. SAT-based Boolean matching has been used for technology mapping of specific circuits such as in [21, 22, and 23]. Yu et al. [21] developed a filter-based Boolean matcher with FC-BM and FH-BM that solved NPN equivalence in FPGA, but this approach could only handle 9 variables. Based on functional decomposition operations, Cong and Hwang of [24] exploited a Boolean matching method for LUT-based PLBs. Soeken et al. [25] proposed a heuristic NPN classification approach for large functions using AIGs and LEXSAT. The authors of [26] researched PP Boolean matching based on graphs, simulation-driven and SAT, which implemented PP Boolean equivalence-checking for large-scale circuits. Many studies of Boolean matching have utilized symmetry because it can effectively reduce the search space. The authors of [27] proposed a generalized Boolean symmetry and applied it to PP Boolean
matching for large circuits. Lai et al. [28] proposed Boolean matching with strengthened learning.

III. PRELIMINARIES AND PROBLEM STATEMENT

In this paper, we use $X = \{x_0, x_1, \ldots, x_{n-1}\}$ to express a vector of Boolean variables and $f(X) : B^n \rightarrow B$ to express a single-output completely specified Boolean function.

Definition 1 (Cofactor of Boolean Function): The cofactor of Boolean function $f$ with respect to variable $x_i(\overline{x}_i)$ is expressed as $f_{x_i}(f_{\overline{x}_i})$, which is computed by substituting $x_i(\overline{x}_i)$ with 1 into $f$.

$f_{x_i}(f_{\overline{x}_i})$ is a new Boolean function of $n-1$ variables. Let $|f_{x_i}|$ denote the number of minterms covered by $f_{x_i}$, and $|f_{\overline{x}_i}|$ denote the number of minterms covered by $f_{\overline{x}_i}$.

A cube $b$ is the conjunction of some variables which can be positive or negative. For any cube $b$, $f_b$ is the cofactor of $f$ with respect to $b$, and $|f_b|$ denotes the number of minterms covered by $f_b$.

Lemma 1: Shannon expansion is the identity $f = x_if_{x_i} + \overline{x}_i f_{\overline{x}_i}$ (also referred to as Shannon decomposition).

The equivalence of two Boolean functions can be verified by recursive Shannon decomposition.

Definition 2 (NP Transformation): An NP transformation $T$ is composed of the negation and/or permutation of input variables. Suppose a Boolean variable is $X = \{x_0, x_1, \ldots, x_{n-1}\}$; its NP transformation can be expressed as $T = \left(\begin{array}{c} x_{s(0)} \cdots x_{s(n-1)} \\ x_{s(1)} \cdots x_{s(n-1)} \end{array}\right)$, where $s$ is a permutation of $\{0, 1, \ldots, n-1\}$, and $x_{s(i)}$ represents whether $x_i$ takes a negation operation, such as $x_i^1 = x_i$, $x_i^0 = \overline{x}_i$.

Example 1: Assume a Boolean function $f(x_0, x_1, x_2) = x_0\overline{x}_1 x_2 + x_0 x_1 \overline{x}_2$ and a transformation $T = \left(\begin{array}{c} x_0 \cdots x_2 \\ x_0^2 \cdots x_2^0 \end{array}\right)$. If $g(x_0, x_1, x_2) = f(TX) = \overline{x}_0 x_1 x_2 + x_0 x_1 \overline{x}_2$, then $f$ is NP-equivalent to $g$.

Definition 3 (PNP Equivalence): Two Boolean functions $f(X)$ and $g(X)$ are PNP equivalent if there exists an NP transformation $T$ that satisfies $f(TX) = g(X)$ or $f(TX) = \overline{g(X)}$. When $x_i$ and $x_j$ satisfy only case (1), the two variables have a mapping with identical phase, which is $\varphi_i : x_i \rightarrow x_j$, $\varphi_j : x_j \rightarrow x_i$. We express this variable mapping by $i \rightarrow j - 1$. If $x_i$ and $x_j$ satisfy both case (1) and (2), we need to consider two variable mappings. In contrast, when $x_i$ and $x_j$ do not satisfy either case (1) or case (2), they have no variable mapping in the transformation detection procedure in this paper.

Example 2: Consider the Boolean functions $f$ and $g$ from Example 2 with the same 1st signature vector. Variable $x_1$ of $f$ and variable $x_0$ of $g$ satisfy case (2) with the variable mapping $1 \rightarrow 0 - 1$.

Definition 4 (0th Signature Value): The 0th signature value of function $f$ is the number of its minterms. In this paper, the 0th signature value is expressed as $|f|$ for $f(X)$.

Definition 5 (1st Signature Value): The 1st signature value of function $f$ with respect to $x_i$ is $V_i = |f_{x_i}| + |f_{\overline{x}_i}|$. The 1st signature vector of $f$ is denoted by $V_f = \{(|f_{x_i}|, |f_{\overline{x}_i}|)\}$.

Because the signature is the key characteristic of Boolean functions, having the same signature is often regarded as a necessary condition for NP-equivalence of Boolean functions [3].

Example 3: Consider the Boolean functions $f$ and $g$ from Example 2 with the same 1st signature vector. Variable $x_1$ of $f$ and variable $x_0$ of $g$ satisfy case (2) with the variable mapping $1 \rightarrow 0 - 1$, $x_1$ of $f$ and $x_0$ of $g$ satisfy both cases (1) and (2); therefore, two possible mappings exist: $0 \rightarrow 1 - 1$ and $0 \rightarrow 1 - 0$.

Definition 6 (Variable-Mapping Set): Every variable has zero or more variable mappings. The variable-mapping set of $x_i$ is the collection that includes all possible variable mappings of $x_i$. The variable-mapping set of $x_i$ is denoted by $\chi_i = \{\varphi_i : x_i \rightarrow x_j^a| j \in \{0, 1, \ldots, n-1\}, a \in \{0, 1\}\}$. In this paper, the variable-mapping set of $x_i$ is simplified to $\chi_i = \{i \rightarrow j - k| j \in \{0, 1, \ldots, n-1\}, k \in \{0, 1\}\}$. An $n$-variable Boolean function has $n$ variable-mapping sets.

The cardinality of the variable-mapping set with respect to $x_i$ is denoted as $|\chi_i|$. When $|\chi_i| = 1$, the variable-mapping set of $x_i$ is called a single-mapping set, and when $|\chi_i| > 1$, it is called a multiple-mapping set. In Example 3, the variable-
mapping set of \( x_0 \) is a multiple-mapping set because \( \chi_0 = \{0 \rightarrow 1 \rightarrow 0, 0 \rightarrow 1 \rightarrow 0, 2 \rightarrow 0, 0 \rightarrow 2 \rightarrow 1 \} \).

**Definition 8 (Minimum variable-mapping set):** An \( n \)-variable Boolean function has \( n \) variable-mapping sets. The minimum variable-mapping set has the smallest cardinality.

If \( |x_i| = 1 \), \( x_i \) has only one variable mapping it means \( x_i \) is distinguished, and when more than one variable exists that satisfies \( |x_i| = 1 \), we can distinguish multiple variables in only one variable-mapping search operation. When there are multiple minimum variable sets and their cardinality is greater than one, the algorithm selects the first minimum variable set to test. In Example 3, the cardinality of the variable-mapping sets of variable \( x_0 \) and \( x_2 \) in \( f \) is 4, and the cardinality of the variable-mapping set of function \( f \) is \( \chi_1 = \{1 \rightarrow 0 \rightarrow 1 \} \).

**Definition 9 (Variable Symmetry):** If an \( n \)-variable Boolean function \( f \) is invariant when variables \( x_i \) and \( x_j (\pi_j) \) are swapped, variables \( x_i \) and \( x_j (\pi_j) \) are symmetric.

Boolean functions can have skew and nonskew symmetries [29]. In this paper, we consider only nonskew symmetry because discovering this symmetry is inexpensive, which reduces the complexity of Boolean matching. Given a pair of variables \( x_i \) and \( x_j \), their cofactors can be used to check the symmetry. If \( f_{x_i} \oplus f_{x_j} = 0 \) (\( f_{x_i} \oplus f_{x_j} = 0 \)), variables \( x_i \) and \( x_j (\pi_j) \) are symmetric. We use this method to discover variable symmetry in our algorithm. When the algorithm detects symmetry, it checks only the variables that have the same 1\(^ st \) signature value. In this paper, we classify variables into two types: symmetric and asymmetric.

**IV. MATCHING ALGORITHM**

The core idea of the algorithm in this paper is to use SS vector and Shannon decomposition to search variable mappings and form possible transformations. The algorithm is based on a fundamental strategy: recursive decomposition and searching. In this section, we describe the entire process of Boolean matching.

**A. Phase Assignment**

In our algorithm, the phase of a Boolean function and its variables must first be determined. We use the following method to assign the phases of the function and its variables.

When two \( n \)-variable Boolean functions, \( f \) and \( g \), are NPN Boolean equivalent, they must satisfy at least one of the two cases: (1) \( |f| = |g| \), (2) \( |f| = 2^n - |g| \); otherwise, they are not NPN Boolean equivalent. If they satisfy only case (1), they may be NP Boolean equivalent and have identical phases; therefore, we assign them to positive phases. If they satisfy only case (2), they may be NPN Boolean equivalent and have opposite phases. In this case, we assign \( f \) to a positive phase and \( g \) to a negative phase. If they satisfy cases (1) and (2) simultaneously, we assign \( f \) to a positive and \( g \) to undetermined because, for \( g \), we need to consider both the positive and negative phases.

**Example 4:** Consider two Boolean functions \( f \) and \( g 

\[ f(X) = x_0x_1x_3 + x_0x_1x_3 + x_0x_1x_2x_3 + x_0x_1x_2x_3 + x_0x_1x_3 \]

\[ g(X) = x_0x_1x_3 + x_0x_1x_2x_3 + x_0x_1x_2x_3 + x_0x_1x_3 + x_0x_1x_2x_3 \]

Because \( |f| = 8 \) and \( |g| = 8 \), the two Boolean functions satisfy both case (1) and (2). We assign a positive phase to both \( f \) and \( g \). The algorithm checks whether \( f \) and \( g \) are NP-equivalent. If they are NP-equivalent, there is no need to test the negative phase of \( g \).

In the procedure of searching for variable mappings, the phases of all the variables of the two Boolean functions must be determined. The phase of variable \( x_i \) is obtained by comparing the relation of \( |f_{x_i}| \) and \( |f_{\pi_i}| \). If \( |f_{x_i}| > |f_{\pi_i}| \), the phase of \( x_i \) is positive. If \( |f_{x_i}| < |f_{\pi_i}| \), the phase of \( x_i \) is negative. If \( |f_{x_i}| = |f_{\pi_i}| \), the phase of \( x_i \) is undetermined. We use 0, 1, and -1 to denote a positive, negative and undetermined phase, respectively.

In Example 4, the 1\(^{st} \) signature vectors of two functions are \( V_f = \{4, 4, 4, 4, 4, 4, 5, 3\} \) and \( V_g = \{3, 5, 4, 4, 4, 4, 4, 4\} \). The phase set of the variables of \( f \) is \( \text{Phase}_f = \{-1, -1, -1, 0\} \). The phase set of the variables of \( g \) is \( \text{Phase}_g = \{1, -1, -1, -1\} \).

**B. Establishing Variable Mappings**

To perform NPN matching of an \( n \)-variable Boolean function, each transformation has \( n \) variable mappings. Searching for variable mappings is critical. Section III introduces the 1\(^{st} \) signature vector. Given two Boolean functions, we can obtain the variable mappings by comparing their 1\(^{st} \) signature values. We classify variable-mapping sets into single-and multiple-mapping sets.

The elementary principle of a variable mapping existing between two variables is that they will have the same 1\(^{st} \) signature value. Although some variables that have the same 1\(^{st} \) signature value are of different variable types (symmetric and asymmetric), NP transformation does not change the variable type; therefore, there is no variable mapping between symmetric and asymmetric variables. Consequently, symmetry information must be added to the 1\(^{st} \) signature value.

In our algorithm, we use \( C_i \) to denote a symmetry class, and \( i \) is the sequence number of the first variable in the symmetry class. We denote the cardinality of each symmetry class by \( |C_i| \). Because two symmetry classes have a mapping if and only if they have the same cardinality and the same 1\(^{st} \) signature value, we add two symmetry marks to the 1\(^{st} \) signature value. One is the cardinality of the symmetry class, the other is the sequence number of the first variable in the symmetry class. If a variable is asymmetric, its two symmetry marks are set to -1.

In Example 4, both \( f \) and \( g \) have one symmetry class. The 1\(^{st} \) signature vectors that include the symmetry information of \( f \) and \( g \) are \( V_f = \{(4, 4, 4, 0), (4, 4, 2, 0), (4, 4, 2, 1), (4, 4, 2, 1), (4, 4, 2, 1)\} \) and \( V_g = \{(3, 5, -1, -1), (4, 4, 2, 1), (4, 4, -1, -1), (4, 4, 2, 1)\} \). Boolean function \( f \) has one symmetry class \( C_0 = \{x_0, x_1\} \), and \( g \) has one symmetry class \( C_1 = \{x_1, x_3\} \).

**Definition 10 (Symmetry Mapping):** When two symmetry classes have the same 1\(^{st} \) signature value and cardinality, they have the mapping \( \varphi : C_i \rightarrow C_j, i, j \in \{0, 1, \cdots, n - 2\} \), and can be written as \( i \rightarrow j \) for short.

If \( f \) has a symmetry class \( C_{i_1} = \{x_{i_1}, x_{i_2}, \cdots, x_{i_m}\} \) and \( g \) has a symmetry class \( C_{j_1} = \{x_{j_1}, x_{j_2}, \cdots, x_{j_m}\} \) and the
two symmetry classes have the same 1st signature value, the variables of \( C_{ij} \) may have a mapping to the variables of \( C_{j1} \). Each variable of \( C_{ij} \) may map to any variable of \( C_{j1} \). Thus, there are \( m! \) possible variable mapping relations between \( C_{ij} \) and \( C_{j1} \). For the invariant property of swapping symmetry variables, these \( m! \) variable mapping relations are equivalent. Consequently, we do not care about the order for establishing variable mappings of two symmetry classes, and only need to establish \( m \) variable mappings \( x_{i_1} \rightarrow y_{j_{1}}, x_{i_2} \rightarrow x_{j_{2}}, \ldots, x_{i_m} \rightarrow x_{j_{m}} \) between \( C_{ij} \) and \( C_{j1} \).

**Definition 11** (symmetry-mapping set): Each symmetry class may have zero or more symmetry mappings. We define the symmetry-mapping set with respect to symmetry class \( C_i \) as \( S_i = \{ \phi_i : C_i \rightarrow C_{ji}, i, j \in \{0, 1, \ldots, n-2\} \} \). In this paper, we simplify the symmetry-mapping set to \( S_i = \{ i \rightarrow j | j \in \{0, 1, \ldots, n-2\} \} \).

Similar to a variable-mapping set, the symmetry-mapping set whose cardinality is one, i.e. \( |S_i| = 1 \), is called a single symmetry-mapping set. When \( |S_i| > 1 \), \( S_i \) is called a multiple symmetry-mapping set. In Example 4, there is a single symmetry-mapping set exists, \( S_0 = \{0 \rightarrow 1\} \).

We can use the methods described above to search for symmetry variable mappings to speed up the variable-mapping search. But there are two problems in the process of variable-mapping search that cannot be solved by symmetry.

(1) By updating the signature, all the 1st signature values of variables may become \((0, 0)\), in which case the variable-mapping search never ends. When this happens, we have no information to distinguish the variables. Of course, we can try to create variable mappings among all the unidentified variables, but this may explode the search space.

(2) After updating 1st signature value, there may be a 1st signature value collision. Some variables that previously had different 1st signature values may have the same 1st signature values after the updating process. When this occurs, we group all the variables of Boolean functions by 1st signature value.

This method avoids searching for incorrect variable mappings.

**Definition 12** (Structural Signature Vector): An \( n \)-variable Boolean function \( f \) has a structural signature vector (SS vector) \( V_f = \{V^0, V^1, \ldots, V^{n-1}\} \). \( V^i \) is the structural signature value of \( x_i \).

SS vector is a new signature vector that adds structural information to the 1st signature vector. The structural signature value of \( x_i \) is \( V^i = (|f_{x_i}|, |f_{\overline{x_i}}|, |C_i|, C_i, G_i) \). It includes a positive variable cofactor, a negative variable cofactor, the cardinality of the symmetry class that \( x_i \) belongs to, the serial number of the first symmetry variable in its symmetry class, and a group serial number.

In our algorithm, we group all the variables. The basic principle of grouping is that variables having the same 1st signature value belong in one group. According to the size of the 1st signature value, we assign a serial number to each group. In Example 4, the SS vectors of \( f \) and \( g \) are

\[
V_f = \{(4,4,2,0,1), (4,4,2,0,0), (4,4,1,1,1), (5,3,1,1,0)\} \quad \text{and} \quad V_g = \{(3,5,1,1,0), (4,4,2,1,1), (4,4,1,1,1), (4,4,2,1,1)\}
\]

For Boolean function \( f \), there are two groups: \( G_0 = \{x_3\} \) and \( G_1 = \{x_0, x_1, x_2\} \).

When the first problem occurs, the algorithm creates variable mappings among the unidentified variables that have the same \( |C_i| \) and \( G_i \). When the second problem occurs, the variables will have different group marks despite having the same 1st signature value. The premise of two variables having a variable mapping is that they will have the same 1st signature value, \( |C_i| \) and \( G_i \).

The symmetry and group marks represent structural information about the Boolean function. When two Boolean functions are NP equivalent, they must have the same 1st signature vector and structure. We first use the SS vector of Boolean functions to estimate the probability of NP equivalence after obtaining the phases of two Boolean functions. When comparing SS vectors, we do not compare \( C_i \) and do not consider order. The SS vector is more efficient than the 1st signature vector. Applying the SS vector reduces the search space and speeds up the matching process. In Example 5, we demonstrate the second problem and show how it is solved.

**C. SS Vector Updating**

There are two cases in which variables cannot be identified:

(1) when more than one variable has the same SS value; and

(2) when the phase of many variables cannot be determined.

The more variables that have the same SS value and undetermined phases, the larger the search space required to perform Boolean matching. Therefore, we utilize the identified variables to update the SS vector, which may change the SS vector and cause unidentified variables to have different SS values. This makes updating the SS vector a critical step. The updating of the SS vector is based on Shannon expansion.

We use splitting variables to decompose the Boolean function. This decomposition forms two new Boolean functions; then, our algorithm computes the SS vector again.

In Example 4, \( f \) has one single-mapping set \( \chi_3 = \{3 \rightarrow 0 \rightarrow 1\} \), one multiple-mapping set \( \chi_2 = \{2 \rightarrow 2 \rightarrow 0 \rightarrow 2 \rightarrow 1\} \), and one single symmetry-mapping set \( S_0 = \{0 \rightarrow 1\} \). Only the phase of variable \( x_3 \) of \( f \) and the phase of variable \( x_0 \) of \( g \) are determined.

Our algorithm first processes variable mapping \( x_3 \rightarrow \overline{x}_0 \). The splitting variable of \( f \) is \( x_3 \) and the splitting variable of \( g \) is \( \overline{x}_0 \). The algorithm presented in this paper uses \( x_3 \) to decompose \( f \) and uses \( \overline{x}_0 \) to decompose \( g \). New SS vectors are obtained by \( x_3f_{x_3} \) and \( \overline{x}_0f_{\overline{x}_0} \). The updated SS vectors are

\[
V_f = \{(3,2,2,0,1), (2,3,2,0,1), (2,3,1,1,1), (0,0,0,1,1), (0,0,1,1,1)\} \quad \text{and} \quad V_g = \{(3,2,2,1,1), (3,2,1,1,1), (2,3,2,1,1)\}
\]

When the first decomposition is complete, the phases of all the variables is determined. Now, the variable-mapping set of \( x_2 \) is \( \{2 \rightarrow 2 \rightarrow 1\} \), and we can obtain two symmetry variable mappings: \( 0 \rightarrow 1 \rightarrow 0 \) and \( 1 \rightarrow 3 \rightarrow 0 \).

However, decomposing the Boolean function and saving the results has considerable space and time costs. In fact, we do not really decompose the Boolean function. We only compute new SS vectors using the splitting variable. Because decomposition may be executed more than once, the splitting
variables gradually form cubes, and we use these cubes to update the SS vectors. Using this approach, the cost of the decomposition procedure is minimal. Our algorithm has two cubes i.e., cube_f and cube_g, where cube_f is the cube of f, and cube_g is the cube of g. When we update the SS vectors, the two cubes will change. cube_f and cube_g are computed by splitting variables as described in Example 5.

algorithm 1 SS Vector Updating

Input: f, g, cube_f, cube_g
Output: 0 or 1

function UPDATE(f, g, cube_f, cube_g)
    Compute_vector(f, cube_f)
    Compute_vector(g, cube_g)
    Update the phase of x in f
    Update the phase of x in g
    Return vector_same(V_f, V_g)
end function

For some Boolean functions, one decomposition may not be sufficient to identify all variables; therefore, they may be decomposed two or more times. In each recursive step, Algorithm 1 decomposes the Boolean function using cube_f and cube_g.

Example 5: Assume we are given two 5-variable Boolean functions:

f(X) = x0x1(x2 + x2x3x4) + x0x1(x3 + x2x3x4 + x2x3x4) + x0x1(x2x3x4 + x2x3x4 + x2x3x4)

and

g(X) = x0x1(x2x3x4 + x2x3x4 + x2x3x4 + x2x3x4) + x0x1(x2x3x4 + x2x3x4 + x2x3x4 + x2x3x4)

(1) In the first recursive call, cube_f = bddtrue, cube_g = bddtrue, and the SS vectors of the two Boolean functions are:

V_f = {{(11, 5, -1, -1, 0), (8, 8, -1, -1, 3), (10, 6, -1, -1, 1), (9, 7, -1, -1, 2), (9, 7, -1, -1, 2)}

V_g = {{(5, 11, -1, -1, 0), (8, 8, -1, -1, 3), (10, 6, -1, -1, 1), (9, 7, -1, -1, 2), (9, 7, -1, -1, 2)}

The two SS vectors are the same, and we obtain the phases of variables of f and g, which are Phase_f = \{0, -1, 0, 0, 0\} and Phase_g = \{-1, 0, 0, 0\}. At this point, the phases of variable x1 of f and g are undetermined. The variables x1 and x3 of function g have different 1st" signature values.

There are two single-mapping sets, \(\chi_0 = \{0 \rightarrow 0 \rightarrow -1\}\) and \(\chi_2 = \{2 \rightarrow 2 \rightarrow -1\}\). We search two variable mappings, which are \(x_0 \rightarrow x_0\) and \(x_2 \rightarrow x_2\), which generates two layers of the transformation tree. The first layer has one variable mapping node \(0 \rightarrow 0 \rightarrow -1\), and the second layer has one variable mapping node \(2 \rightarrow 2 \rightarrow -1\). The splitting variable of \(f\) is \(x_0\), and the splitting variable of \(g\) is \(x_0\). Algorithm 2 updates cube_f and cube_g by cube_f = \(x_0\) and cube_g = \(x_o\).

(2) In the second recursive step, Algorithm 1 utilizes \(x_0\) and \(x_0\) to update the SS vectors of \(f\) and \(g\). The results are as follows:

V_f = {{(0, 0, -1, -1, 0), (5, 6, -1, -1, 3), (0, 0, -1, -1, 1), (6, 5, -1, -1, 2), (6, 5, -1, -1, 2)}

V_g = {{(0, 0, -1, -1, 0), (6, 5, -1, -1, 3), (0, 0, -1, -1, 1), (6, 5, -1, -1, 2), (6, 5, -1, -1, 2)}

Phase_f = \{0, 1, 0, 0, 0\}, Phase_g = \{1, 0, 0, 0, 0\}

From the updated SS vectors, we can obtain the following information:

1) The phases of all variables are determined.
2) The changed SS vectors are conductive to variable identification. We can identify variable \(x_1\) of \(f\) and \(g\). There exists a single-mapping set \(\chi_1 = \{1 \rightarrow 1 \rightarrow 1\}\). The splitting variable of \(f\) and \(g\) is \(x_2\). We can compute cube_f = \(x_0x_2\) and cube_g = \(x_0x_2\) using Algorithm 2.
3) From the changed SS vectors, we can see that variables \(x_1\) and \(x_3\) of \(g\) have the same 1st signature value, which conflicts with the previous result. If there is no group mark, the algorithm will search for an incorrect variable mapping between variable \(x_1\) of \(f\) and \(x_3\) of \(g\), and this incorrect variable mapping will cause one or more incorrect transformations. Thus, incorrect variable mappings increase the search space.

D. Transformation Detection

The main goal of Algorithm 2 is to create an NP transformation that maps \(f\) to \(g\). The basis of transformation detection is to search the variable mappings of all variables. An \(n\)-variable Boolean function has \(n2^n\) NP transformations, which leads to very high complexity, making the exhaustive method suitable only for functions with fewer input variables. However, in reality, the number of NP transformations that transform \(f\) to \(g\) is small.

In the process of transformation detection, Algorithm 2 creates a tree to build all possible transformations. Each unbranched branch is a transformation. This tree has \(n\) layers, and each layer has one or more variable mapping nodes. Algorithm 2 utilizes depth-first search variable mapping to detect transformation. Then, it checks whether \(f(TX) = g(X)\) or \(f(TX) = g(X)\) until either finds a transformation \(T\) that can meet this condition or until no transformations can meet this condition.

In Algorithm 2, conditions \(D_1\), \(D_2\), \(D_3\) and \(D_4\) are as follows:

\(D_1\) occurs when the depth of tree reaches \(n\).
\(D_2\) occurs when the 1st signature values of all variables of \(f\) and \(g\) become \((0, 0)\) but unidentified variables still exist.
\(D_3\) occurs when the single-mapping sets or single symmetry-mapping sets are searched.
\(D_4\) occurs when there are multiple symmetry-mapping sets.

In the transformation detection, Algorithm 2 searches variable mapping for each undetermined variable. And we handle the single-mapping set and single symmetry-mapping set condition first. As soon as such mapping sets are found, we create corresponding mapping nodes and add them to the tree. When no single-mapping set and single symmetry-mapping set exist, Algorithm 2 first handles the multiple symmetry-mapping set and then handles the multiple-mapping set.

During each recursion, Algorithm 2 identifies one or more variables and adds one or more variable mapping nodes to the transformation tree. When more than one single-mapping sets or single symmetry-mapping sets exist, the depth of the transformation tree will be increased more than once. After the addition of variable mapping nodes, Algorithm 2 updates cube_f and cube_g. In Algorithm 2, the cube is not updated when handling the multiple symmetry-mapping set. When
Algorithm 2 Transformation Detection

Input: $f, g, cube_f, cube_g, map_list$
Output: 0 or 1

function DETECT($f, g, cube_f, cube_g, map_list$)
    if $D_1$ then
        Create a transformation
        Return VERIFY($f, g, map_list$)
    else if UPDATE($f, g, cube_f, cube_g$) = 0 then
        Return 0
    else if $D_2$ then
        Search variable mapping using the symmetry and group marks
        Add Map_node to Map_list
        Return DETECT($f, g, cube_f, cube_g, map_list$)
    else
        for all $x_i \in f(x)$ do
            Compute $|x|$ if $|x_i| = 1 or |S_i| = 1$ then
                Create Map_node for $x_i$ or $C_i$
                Add Map_node to Map_list
            else
                Get Min(|$x_i$|)
            end if
        end for
        if $D_3$ then
            Update $cube_f$ and $cube_g$
            Return DETECT($f, g, cube_f, cube_g, map_list$)
        else if $D_4$ then
            Search the variable mappings in a multiple symmetry-mapping set
            Add Map_node to Map_list
            Return DETECT($f, g, cube_f, cube_g, map_list$)
        else
            Create Multiple Map_node_list
            Select one Map_node as Current Node
            ADD Current Map_node to Map_list
            Update $cube_f$ and $cube_g$
            Return DETECT($f, g, cube_f, cube_g, map_list$)
        end if
    end if
end function

$D_1$ is true, a transformation is generated. When $D_1$ is false, Algorithm 2 calls Algorithm 1 and Algorithm 1 updates the SS vectors according to $cube_f$ and $cube_g$, and then returns to the variable-mapping search.

When a layer of the transformation tree has more than one node, there may be incorrect variable mappings. These incorrect mappings generate incorrect branches, which increase the search pace and slow down matching. Therefore, we must find incorrect variable mappings and stop these branches from being created as soon as possible. We terminate the current branch in the following two cases:

1. when the two updated SS vectors are not equal; and
2. when a new variable mapping introduces phase collision.

Phase collision is a change of the phase relation between two variables. When a multiple-mapping set is found, there are multiple variable mappings that can be selected. Algorithm 2 selects the variable mappings in sequent, which may lead to phase collision.

Suppose the minimum variable-mapping set is $\chi_3 = \{j_1 \rightarrow k_1, \ldots, j_m \rightarrow k_m\}$, Algorithm 2 selects $i \rightarrow j_1 \rightarrow k_1$ as the current variable mapping and generates the corresponding variable mapping node. Assuming that variables $x_h$ and $x_l$ have the same phase, Algorithm 2 continues to search the next variable mapping.

Algorithm 1 updates the SS vectors and the variable phases. If we find a single mapping $h \rightarrow l - 1$, then $x_h$ and $x_l$ have opposite phases, which conflicts with the previous condition. This is called phase collision. Phase collision is caused by an incorrect variable-mapping selection $i \rightarrow j_1 \rightarrow k_1$.

Example 6: For the Boolean functions in Example 5, after the first decomposition and updating of the SS vectors, three variables are already identified. We can find that variable $x_4$ of $f$ and $g$ have the same phase. Now, $cube_f = x_0 x_2$ and $cube_g = x_0 x_2$.

1. Because the depth of the transformation tree does not reach $n$, it updates the SS vectors. The results are

   $V_f = \{(0, 0, -1, -1, 0), (0, 0, -1, -1, 3), (0, 0, -1, -1, 1), (3, 3, -1, -1, 2), (3, 3, -1, -1, 2)\}$

   $V_g = \{(0, 0, -1, -1, 0), (0, 0, -1, -1, 3), (0, 0, -1, -1, 1), (3, 3, -1, -1, 2), (3, 3, -1, -1, 2)\}$

The two SS vectors above have two minimum variable sets: $\chi_3 = \{3 \rightarrow 3 - 0, 3 \rightarrow 4 - 0\}$ and $\chi_4 = \{4 \rightarrow 3 - 0, 4 \rightarrow 4 - 0\}$. Algorithm 2 selects the first minimum variable set $\chi_3$. The transformation tree is extended by one layer, with two variable nodes $3 \rightarrow 3 - 0$ and $3 \rightarrow 4 - 0$. This minimum mapping set has two variable mappings. We select $3 \rightarrow 3 - 0$ as the current variable mapping. Algorithm 2 updates $cube_f$ and $cube_g$ with $cube_f = x_0 x_2 \tau_1$ and $cube_g = x_0 x_2 \tau_1$.

2. Algorithm 1 updates the SS vectors, and the results are:

   $V_f = \{(0, 0, -1, -1, 0), (0, 0, -1, -1, 3), (0, 0, -1, -1, 1), (0, 0, -1, -1, 2), (1, 2, -1, -1, 2)\}$

   $V_g = \{(0, 0, -1, -1, 0), (0, 0, -1, -1, 3), (0, 0, -1, -1, 1), (0, 0, -1, -1, 2), (2, 1, -1, -1, 2)\}$

From the above SS vector results, we know variable $x_4$ of $f$ and $g$ have opposite phases. But they previously had the same phase; therefore, we have found a phase collision. The reason this collision occurs is the selection of the variable mapping $3 \rightarrow 3 - 0$. Therefore, this branch is terminated. Algorithm 2 returns $3 \rightarrow 4 - 0$ and continues to search the rest of the variable mappings.

From Example 6, we see that our algorithm finds incorrect variable mappings quickly. If we did not check for phase collision, the wrong branch would continue to execute and might even verify that $f(TX) = g(X)$ or $f(TX) = g(X)$. Phase collision checking accelerates the algorithm to some extent.

Example 7: Consider the two 7-variable Boolean functions $f(X)$ and $g(X)$.

$f(X) = x_0 x_1 x_2 x_3 x_4 + x_0 x_1 x_2 x_3 x_5 + x_0 x_1 x_2 (x_3 x_5 x_6 + x_3 x_5 x_6) + x_0 x_1 x_2 x_4 x_5 + x_0 x_1 x_2 (x_3 x_5 x_6 + x_4 x_5 x_6) + x_0 x_1 x_2 (x_3 x_5 x_6 + x_4 x_5 x_6) + x_0 x_1 x_2 x_4 x_5 + x_0 x_1 x_2 x_4 x_5 + x_0 x_1 x_2 x_4 x_5 + x_0 x_1 x_2 x_4 x_5$

$g(X) = x_0 x_1 x_2 (x_3 x_5 x_6 + x_3 x_5 x_6) + x_0 x_1 x_2 x_4 x_5 + x_0 x_1 x_2 x_4 x_5 + x_0 x_1 x_2 x_4 x_5 + x_0 x_1 x_2 x_4 x_5$

(1) The initial values of $cube_f$ and $cube_g$ are both true. The depth of the transformation tree is 0. Condition $D_1$ is false. Algorithm 2 computes the SS vectors of $f$ and $g$. The results are:
The mapping set has been searched. Algorithm 2 searches the minimum variable-mapping set \( \chi_2 = \{2 \rightarrow 5 \rightarrow 1\} \). This is a single-mapping set. Algorithm 2 adds variable mapping \( x_2 \rightarrow \tau_5 \) to the transformation tree. The first layer of the transformation tree has one mapping node. The splitting variables of \( f(X) \) and \( g(X) \) are \( x_2 \) and \( \tau_5 \), respectively. Algorithm 2 separately updates \( cube_f \) and \( cube_g \) with \( cube_f = x_2 \) and \( cube_g = \tau_5 \), and then proceeds with the next recursive call.

(2) Condition \( D_1 \) is false. Algorithm 1 updates the SS vectors. The results are:

\[
V_f = \{(19, 12, 2, 0, 1), (19, 12, 2, 1, 1), (0, 0, -1, -1, 0), (12, 19, 2, 1, 1), (19, 12, 2, 0, 1), (20, 11, -1, -1, 2), (11, 20, -1, -1, 2)\}
\]

\[
V_g = \{(0, 0, 2, 0, 1), (11, 20, -1, -1, 2), (12, 19, 2, 0, 1), (19, 12, 2, 3, 1), (19, 12, 2, 3, 1), (0, 0, -1, -1, 0), (20, 11, -1, -1, 2)\}
\]

These two SS vectors are the same and condition \( D_2 \) is false, so Algorithm 2 searches the minimum variable-mapping set. Condition \( D_3 \) is false and \( D_1 \) is true. The algorithm finds two minimum symmetry-mapping sets: \( S_0 = \{0 \rightarrow 0, 0 \rightarrow 3\} \) and \( S_1 = \{1 \rightarrow 0, 1 \rightarrow 3\} \). Algorithm 2 selects the first minimum symmetry-mapping set \( S_0 \). Two branches are generated by the symmetry mappings, 0 to 0 and 0 to 3. Algorithm 2 selects symmetry mapping 0 -> 0, and generates two variable mappings \( 0 \rightarrow 0 - 1 \) \((x_0 \rightarrow \tau_0)\) and \( 4 \rightarrow 2 - 1 \) \((x_4 \rightarrow \tau_2)\). Then, it adds two layers to the transformation tree, each of which has one variable mapping node. If the transformations created by symmetry mapping \( 0 \rightarrow 0 \) are verified to be false, Algorithm 2 returns and selects \( 0 \rightarrow 3 \) to handle. Then, it continues with the next recursive call.

(3) Condition \( D_4 \) is false. Algorithm 1 updates the SS vectors and the results are

\[
V_f = \{(0, 0, 2, 0, 1), (19, 12, 2, 1, 1), (0, 0, -1, -1, 0), (12, 19, 2, 1, 1), (0, 0, 2, 0, 1), (20, 11, -1, -1, 2), (11, 20, -1, -1, 2)\}
\]

\[
V_g = \{(0, 0, 2, 0, 1), (11, 20, -1, -1, 2), (0, 0, 2, 0, 1), (19, 12, 2, 3, 1), (19, 12, 2, 3, 1), (0, 0, -1, -1, 0), (20, 11, -1, -1, 2)\}
\]

Because condition \( D_2 \) is false, the minimum variable-mapping set has been searched. Algorithm 2 searches the single symmetry-mapping set \( S_1 = \{1 \rightarrow 3\} \). It forms two variable mappings: \( 1 \rightarrow 3 \rightarrow 0 \) \((x_1 \rightarrow x_3)\) and \( 3 \rightarrow 4 \rightarrow 1 \) \((x_3 \rightarrow x_4)\). The depth of the transformation tree becomes 5. Algorithm 2 updates \( cube_f \) and \( cube_g \) with \( cube_f = x_0\tau_2 \) and \( cube_g = \tau_5\tau_0 \) and then continues with the next recursive call.

(4) This step is same as the last step. Algorithm 1 updates the SS vectors:

\[
V_f = \{(0, 0, 2, 0, 1), (0, 0, 2, 1, 1), (0, 0, -1, -1, 0), (0, 0, 2, 1, 1), (0, 0, 2, 0, 1), (10, 9, -1, -1, 2), (7, 12, -1, -1, 2)\}
\]

\[
V_g = \{(0, 0, 2, 0, 1), (7, 12, -1, -1, 2), (0, 0, 2, 0, 1), (0, 0, 2, 3, 1), (0, 0, -1, -1, 0), (10, 9, -1, -1, 2)\}
\]

Condition \( D_2 \) is false and there are two single-mapping sets \( \chi_5 = \{5 \rightarrow 6 \rightarrow 0\} \) and \( \chi_6 = \{6 \rightarrow 1 \rightarrow 0\} \); therefore, Algorithm 2 adds two layers to the transformation tree for variable mapping nodes \( x_5 \rightarrow x_6 \) and \( \tau_6 \rightarrow \tau_1 \). \( cube_f \) and \( cube_g \) are updated with \( cube_f = x_0\tau_2 \tau_4 \) and \( cube_g = \tau_5\tau_0\tau_2 \). Then, Algorithm 2 continues with the next recursive call.

(5) Condition \( D_1 \) is true. Algorithm 2 creates a transformation \( T = \{2 \rightarrow 5 \rightarrow 1 \rightarrow 0 \rightarrow 1 \rightarrow 4 \rightarrow 2 \rightarrow 1 \rightarrow 3 \rightarrow 0 \rightarrow 3 \rightarrow 4 \rightarrow 1 \rightarrow 5 \rightarrow 6 \rightarrow 0 \rightarrow 1 \} \).

**E. Boolean Matching**

There are multiple transformations between two NPN equivalent Boolean functions. We need to find only one transformation that can transform \( f \) to \( g(\overline{g}) \) to prove that \( f \) is equivalent to \( g(\overline{g}) \). We explained the process of transformation detection in part D. The two Boolean functions are equivalent when calling DETECT() returns 1; otherwise they are non-equivalent. Our algorithm performs some initialization work for \( f \) and \( g \) such as creating the BDD of \( f \) and \( g \), determining their phases, computing their SS vectors, and so on. Algorithm 3 calls DETECT() when \( f \) and \( g \) satisfy (1) \(|f|=|g| \) or \(|f|=|g|\) and (2) \( V_f = V_g \) or \( V_f = V_7 \); otherwise they are not equivalent. These ideas are summarized in Algorithm 3.

**Theorem 1.** Suppose the Boolean functions \( f(X) \) and \( g(Y) \) are NPN equivalent. Our algorithm must be able to find an NP transformation \( T = \{i \rightarrow j - k| i, j \in \{0, 1, \cdots, n-1\}, k \in \{0, 1\}\} \) that satisfies \( f(TX) = g(X) \) or \( f(TX) = g(X) \).

**Proof:**

Because Boolean function \( f \) is equivalent to \( g \), they must satisfy:

(i) \(|f|=|g| \) or \(|f|=2^n - |g|\).

(ii) \( V_f = V_g \) or \( V_f = V_7 \).

Suppose two NP-equivalent Boolean functions \( f \) and \( g(\overline{g}) \) have an NP transformation \( T \). Then, there must be a variable mapping \( x_i \rightarrow x_j(\overline{\tau}) \) in \( T \), and this variable mapping must satisfy one of the following conditions: (iii) or (iv).

(iii) If variable \( x_i \) is an asymmetrical variable, variable \( x_j \) must be an asymmetrical variable and it must have the same 1st signature value as variable \( x_i \).

(iv) If variable \( x_i \) is a symmetry variable and it belongs to a symmetry class with cardinality \( m \), variable \( x_j \) must be a symmetry variable, the cardinality of its symmetry class must be \( m \), and its 1st signature value must be the same as that of \( x_i \).

Algorithm 3 ensures (i) and (ii). During the transformation detection, Algorithm 2 searches and handles single-mapping sets, multiple-mapping sets, single symmetry-mapping sets and multiple symmetry-mapping sets. The variable mapping of single-mapping sets and multiple-mapping sets belongs to (iii). The variable mapping of single symmetry-mapping sets and multiple symmetry-mapping sets belongs to (iv).
Algorithm 3 Initialization

\[\text{cube}_f = \text{bddtrue}, \text{cube}_g = \text{bddtrue}, \text{map}_\text{list} = \text{NULL}\]

Create Boolean function \(f\) and compute \(|f|, V_f\)
Create Boolean function \(g\) and compute \(|g|\)
if \(|f| = |g|\) then
if \(|f| \neq 2^n - 1\) then
Compute \(V_g\)
if \(V_f = V_g \land \text{DETECT}(f, g, \text{cube}_f, \text{cube}_g, \text{map}_\text{list}) = 1\) then
\(f\) is equivalent to \(g\)
else
\(f\) is not equivalent to \(g\)
end if
else
Compute \(V_g\)
if \(V_f = V_g \land \text{DETECT}(f, g, \text{cube}_f, \text{cube}_g, \text{map}_\text{list}) = 1\) then
\(f\) is equivalent to \(g\)
else
Compute \(V_g\)
if \(V_f = V_g \land \text{DETECT}(f, g, \text{cube}_f, \text{cube}_g, \text{map}_\text{list}) = 1\) then
\(f\) is equivalent to \(g\)
else
\(f\) is not equivalent to \(g\)
end if
else
Compute \(V_g\)
if \(V_f = V_g \land \text{DETECT}(f, g, \text{cube}_f, \text{cube}_g, \text{map}_\text{list}) = 1\) then
\(f\) is equivalent to \(g\)
else
\(f\) is not equivalent to \(g\)
end if
else
\(f\) is not equivalent to \(g\)
end if

Algorithm 2 searches the variable-mapping set for each variable. After a single-mapping set or single symmetry-mapping set has been searched, Algorithm 2 creates mapping nodes and adds them to the transformation tree immediately.

After all variables have been searched, if there are no single-mapping sets and single symmetry-mapping sets, there must be a multiple symmetry-mapping set or a multiple-mapping set. Algorithm 2 processes the multiple symmetry-mapping set first. Let the minimum multiple symmetry-mapping set be \(\{i \rightarrow j_1, i \rightarrow j_2, \ldots, i \rightarrow j_m | j_1, j_2, \ldots, j_m \in \{0, 1, \ldots, n - 1\}\}\) and each symmetry class have \(m\)1 variables. Algorithm 2 first selects the symmetry mapping \(i \rightarrow j_1\). Then, it creates \(m\)1 mapping nodes and adds them to the transformation tree. If the transformation generated by \(i \rightarrow j_1\) is verified to be false, Algorithm 2 will return and select one of the remaining symmetry mappings from the minimum symmetry-mapping set until one is verified to be correct or all have been verified as false.

If there is no single-mapping set, single symmetry-mapping set, or multiple symmetry-mapping set, there must be multiple-mapping sets. Algorithm 2 selects and handles the minimum multiple-mapping set. Let the minimum multiple-mapping set have \(k\) variable mappings; the algorithm selects the first one to handle. It creates a mapping node and adds it to the transformation tree. The method for processing a multiple-mapping set is similar to that used for the multiple symmetry-mapping set. If the transformation generated by the first variable mapping in the minimum multiple-mapping set is verified to be false, the algorithm returns and selects one of the remaining variable mappings from the minimum multiple-mapping set until one is verified to be correct or all have been verified as false.

When the 1st signature values of all variables becomes \((0, 0)\) and some variables have not been identified, Algorithm 2 searches the variable-mapping set using the symmetry and group marks. The method for processing variable mappings is the same as described above.

During transformation detection, the variable mappings of transformation found by Algorithm 2 meet (iii) and (iv). For the variable mappings of multiple-mapping sets and multiple symmetry-mapping sets, our algorithm may traverse each variable mapping. Therefore, if \(f\) is NPN equivalent to \(g\), our algorithm will be able to find a transformation that can transform \(f\) to \(g\).

During the process of transformation detection, each unabridged branch of the transformation tree is an NP transformation detection. Considering the Boolean matching of the Example 7, there are \(7!2^7\) possible transformations using exhaustive method. In contrast, using our proposed algorithm, there are only 2 possible transformations. All the possible transformations are shown in Fig. 1.

![Fig. 1. Transformation Search Tree for Example 7](image)

Because Algorithm 2 uses the depth-first search method, it does not need to search and verify all the transformations. In the execution of Example 7, the algorithm returns 1 when Algorithm 2 detects the first transformation. Fig. 2 shows the actual transformation search tree of Example 7.
From Example 7, using the SS vector and Shannon expansion greatly reduces the search space. Using the depth-first search method further improves the matching speed.

V. EXPERIMENTAL RESULTS

To demonstrate the effectiveness of our proposed algorithm, we conducted an experiment using large circuit sets. We executed our algorithm on the input sets that contains NPN equivalent as well as NPN non-equivalent circuits. We tested the algorithm using both MCNC benchmark circuits and randomly generated circuits. There are two types of randomly generated circuits. In the first type, the circuit is generated by a random integer and the number of minterms of these circuit functions is random. In the second type, the circuit is randomly generated by a string of 0s and 1s whose circuit function has $2^{n-1}$ minterms. For the non-equivalence testing experiment, two Boolean functions $f$ and $g$ have the same or complementary $0^t$ signature values. We tested input Boolean functions with 7-22 variables. For each circuit function obtained from the MCNC benchmark or random circuit set, we randomly created 50 NPN transformations and the corresponding equivalent circuit functions. The numbers of different input circuits taken from the MCNC benchmark are different. For example, we selected 28 19-input circuits and 40 20-input circuits from the MCNC benchmark. For the random circuit set, we randomly generated 50 circuit functions for different input circuits.

In this paper, we compared the results of our algorithm with [6]. The authors of [6] proposed an algorithm for computing canonical form in which two Boolean functions are equivalent if they have identical canonical forms. Because [6] did not report the runtimes for Boolean matching, we re-implemented the algorithm in [6] using the Boolean function set generated from MCNC benchmark and tested to ensure that the results for computing canonical forms was consistent with the result of [6] under a comparable hardware environment. Then, we compared these two algorithms in a new hardware environment with a 3.3 GHz CPU and 4GB RAM. The runtime we report is the CPU time used. The runtimes are reported in seconds and include Boolean functions with up to 22 inputs. We report the minimum, maximum and average runtimes of the algorithms proposed in this paper and the algorithm from [6], respectively. In the following tables, the first column lists the number of input variables (#I) followed by three columns that show the minimum (#MIN), maximum (#MAX) and average (#AVG) of runtime. The next three columns are the corresponding runtimes of the algorithm from [6]. The last columns of Tables I, II and III list the average BDD size (#AVG nodes).

Table I shows the results on the equivalent MCNC benchmark circuits. These Boolean functions are pseudorandomly generated from a subject circuit in the MCNC benchmark circuits. Fig. 3 shows the average runtime of our algorithm compared to the average runtime of [6] on equivalent MCNC benchmark circuits. Table II shows the results from testing the first type of equivalent random circuits, and Fig 4 shows the average runtime of our algorithm compared to that of [6] on the first type of equivalent random circuits.

### Table I

| #I | #MIN | #MAX | #AVG | #MIN of Ref[6] | #MAX of Ref[6] | #AVG of Ref[6] | #AVG nodes |
|----|------|------|------|----------------|----------------|----------------|------------|
| 7  | 0.00002 | 0.00035 | 0.00011 | 0.000045 | 0.05302 | 0.00121 | 19108 |
| 8  | 0.00013 | 0.00059 | 0.00030 | 0.00008 | 0.01164 | 0.00122 | 30616 |
| 9  | 0.00005 | 0.00111 | 0.00044 | 0.00011 | 0.00318 | 0.00186 | 41391 |
| 10 | 0.00011 | 0.00185 | 0.00065 | 0.00020 | 0.00245 | 0.00193 | 170847 |
| 11 | 0.00009 | 0.00217 | 0.00078 | 0.00027 | 0.00505 | 0.00224 | 124842 |
| 12 | 0.00018 | 0.00625 | 0.00101 | 0.00025 | 0.01480 | 0.00255 | 282490 |
| 13 | 0.00089 | 0.02500 | 0.00277 | 0.00126 | 0.19924 | 0.00553 | 554213 |
| 14 | 0.00073 | 0.01809 | 0.00531 | 0.00112 | 0.03291 | 0.01245 | 340672 |
| 15 | 0.00039 | 0.05908 | 0.00755 | 0.00102 | 0.14415 | 0.04077 | 374421 |
| 16 | 0.00064 | 0.31375 | 0.02029 | 0.00074 | 0.06281 | 0.04849 | 604159 |
| 17 | 0.00102 | 0.65791 | 0.14711 | 0.00067 | 1.19085 | 0.31644 | 345396 |
| 18 | 0.00192 | 1.63740 | 0.16584 | 0.00132 | 4.10735 | 0.64273 | 360014 |
| 19 | 0.00391 | 1.94918 | 0.88743 | 0.18556 | 5.85315 | 1.46287 | 308160 |
| 20 | 0.00904 | 4.75490 | 1.96381 | 0.18556 | 10.44006 | 2.13027 | 562698 |
| 21 | 0.16699 | 5.74320 | 4.14688 | 0.29532 | 29.25150 | 10.21228 | 523303 |
| 22 | 1.29078 | 16.44130 | 6.24368 | 1.24569 | 30.36820 | 11.27597 | 621788 |

Fig. 3. Comparison of average runtimes on equivalent MCNC benchmark circuits

We can see that the matching speed in Table II is faster than that shown in Table I. This result occurs because the experiment on the first type of equivalent random circuits generates fewer multiple-mapping sets than that on the
equivalent MCNC benchmark circuits. Greater numbers of multiple-mapping sets and larger cardinality will generate more branches; consequently, Algorithm 2 must test \( f(TX) = g(X) \) or \( f(TX) = \overline{g(X)} \) many more times.

Table III shows the results tested on the second type of equivalent random circuits. Fig. 5 shows the average runtime of our algorithm compared to the average runtime of Ref[6] on the second type of equivalent random circuits.

The number of minterms in the second type of random circuits is \( 2^{n-1} \). Our algorithm first matches \( f \) and \( g \) and then matches \( f \) and \( \overline{g} \) when \( f \) and \( g \) are not NP-equivalent. Compared with Tables I and II, the runtime of Table III is significantly longer than those of Tables I and II. This result occurs because transformation detection is more likely to execute twice.

From Tables I, II and III, we can see our algorithm is faster than Ref[6]. On average, our runtime is generally approximately half that of Ref[6] when tested on equivalent functions. Nevertheless, our algorithm is slower than Ref[6] in the worst case. In this case, the \( j^{th} \) signature value of circuit functions is \( 2^{n-1} \), and the \( 1^{st} \) signature value of multiple variables is \( 2^{n-2} \). This is because the first decomposition, \( 2^{n-3} \), after the first decomposition, and there are no symmetric variables. At this point, the number of possible transformations increases exponentially with the number of inputs, and Algorithm 2 must check \( f(TX) = g(X)(\overline{g(X)}) \) many times, which wastes a lot of time. However, the execution speed of our algorithm is generally linear with the number of inputs on the general circuits. Overall, our algorithm is superior for the majority of circuits with the exception of the worst case when matching equivalent circuit functions.

We also tested our algorithm on the non-equivalent circuits. Table IV shows the experimental results on the non-equivalent MCNC benchmark circuits. Fig. 6 shows the average runtime of our algorithm compared with the average runtime of Ref[6] on the non-equivalent MCNC benchmark circuits. Table V lists...
the experimental results on non-equivalent random circuits, and Fig. 7 shows the average runtime of our algorithm compared with the average runtime of [6] on non-equivalent random circuits.

### TABLE IV

**Boolean matching runtimes on the non-equivalent MCNC benchmark circuits**

| #I  | #MIN | #MAX | #AVG | #MIN of Ref[6] | #MAX of Ref[6] | #AVG of Ref[6] |
|-----|------|------|------|---------------|---------------|---------------|
| 7   | 0.00001 | 0.00024 | 0.00005 | 0.00004 | 0.00048 | 0.00010 |
| 8   | 0.00003 | 0.00024 | 0.00010 | 0.00028 | 0.00061 | 0.00125 |
| 9   | 0.00004 | 0.00026 | 0.00011 | 0.00046 | 0.00024 | 0.00145 |
| 10  | 0.00004 | 0.00046 | 0.00018 | 0.00026 | 0.00030 | 0.00075 |
| 11  | 0.00005 | 0.00033 | 0.00020 | 0.00030 | 0.00048 | 0.00260 |
| 12  | 0.00006 | 0.00053 | 0.00020 | 0.00030 | 0.00120 | 0.00208 |
| 13  | 0.00023 | 0.00102 | 0.00040 | 0.00111 | 0.10677 | 0.00234 |
| 14  | 0.00020 | 0.00253 | 0.00088 | 0.00135 | 0.04268 | 0.03071 |
| 15  | 0.00019 | 0.00313 | 0.00085 | 0.00076 | 0.11391 | 0.01109 |
| 16  | 0.00011 | 0.00232 | 0.00086 | 0.00080 | 0.00476 | 0.01865 |
| 17  | 0.00007 | 0.00257 | 0.00088 | 0.00071 | 1.05822 | 0.38897 |
| 18  | 0.00006 | 0.00479 | 0.00117 | 0.00124 | 3.29635 | 0.37848 |
| 19  | 0.00012 | 0.01434 | 0.00273 | 0.00304 | 3.76650 | 1.53924 |
| 20  | 0.00066 | 0.00898 | 0.00279 | 0.19141 | 10.11520 | 2.29006 |
| 21  | 0.00091 | 0.00678 | 0.00359 | 0.33115 | 10.36450 | 3.46260 |
| 22  | 0.00113 | 0.02299 | 0.00697 | 9.15070 | 27.38330 | 12.84580 |

As shown in Figs. 6–7 and Tables IV and V, the average runtime of our algorithm is approximately 10 to 20 times faster than that of [6] when the number of inputs is small. As the number of inputs increase, the average runtime of our algorithm becomes far faster than [6]. As Table IV shows, our algorithm is 1,800 times faster than [6] when the number of inputs is 22.

In the process of non-equivalent circuit matching, there are three possible cases. (1) Matching terminates after the first equivalence judgment of two SS vectors; in other words, our algorithm does not have to perform transformation detection. (2) Our algorithm executes transformation detection, but does not find any possible transformations. (3) Our algorithm executes transformation detection and finds some possible transformations, but none of these transformations meet the condition \( f(TX) = g(X) \) or \( f(TX) = g(X) \). In our testing, almost all non-equivalent circuit functions belong to (1) or (2). Our algorithm requires considerable time to generate circuit functions that belong to (3). We generated a small number of such circuit functions and tested them. The runtime of our algorithm for non-equivalent circuit matching is faster than on equivalent circuit matching because the latter have very few possible transformations. In contrast, the algorithm of [6] must compute the canonical form twice anyway. Even when the canonical form of a circuit functions is stored in a cell library in advance, the algorithm of [6] is slower than ours because it must still compute one canonical form. Therefore, our algorithm has an obvious advantage over the algorithm from [6] when it comes to matching non-equivalent Boolean functions.

From the non-equivalent matching results, the performance
of our algorithm is more robust than that of [6]. Specifically, as shown in Table IV, when the number of inputs is 22, the average runtime of our algorithm is approximately 140 times greater than when the number of inputs is 7, while the average runtime of algorithm of [6] is about 1200 times greater than that when the number of inputs is 7.

Using BDD to represent Boolean functions, the space requirement depends on the size of BDD. Theoretically, a Boolean function must be decomposed \( n - 1 \) times, and the time complexity increases exponentially with the number of inputs in extreme cases. Actually, in our algorithm, the decomposition scenario rarely occurs; for most functions, it can determine whether a matching transformation exists with only a few decompositions.

The experiment results illustrate the effectiveness of the proposed algorithm and demonstrate that it can be applied to Boolean matching for large-scale circuits.

VI. CONCLUSIONS

This paper proposes an efficient NPN Boolean matching algorithm based on a new SS vector and the Shannon expansion theorem. The proposed algorithm prunes the search space and significantly reduces the time complexity.

Compared with the algorithm of [6], our algorithm is two times faster on general equivalent circuits, and, on average, at least one hundred times faster on non-equivalent circuits. Our algorithm is far faster on non-equivalent matching and is more robust compared with the algorithm of [6]. The algorithm can be used for both large scale circuit technology mapping and cell-library binding. Our algorithm’s weakness lies in matching for the worst equivalent circuit functions. In future work, we will investigate how to solve this problem and apply our algorithm to Boolean matching with don’t-care sets and to multiple-output Boolean function matching.

ACKNOWLEDGMENT

We would like to thank the National Natural Science Foundation of China (Grant No. 61572109) for the technology support.

REFERENCES

[1] G. Agosta, F. Bruschi, G. Pelosi, and D. Sciuto, “A transform-parametric approach to boolean matching,” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 28, no. 6, pp. 805–817, 2009.

[2] K. C. Chen and J. C. Y. Yang, “Boolean matching algorithms,” in International Symposium on VLSI Technology, Systems, and Applications, Proceedings, Taipei, Taiwan, May 1993, pp. 44–48.

[3] A. Abdollahi, “Signature based boolean matching in the presence of don’t cares,” in Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE, Anaheim, CA, 2008, pp. 642–647.

[4] B. Kapoor, “Improved technology mapping using a new approach to boolean matching,” in European Design and Test Conference, Proceedings, Paris, Mar. 1995, pp. 86–90.

[5] M. Damiani and A. Y. Selchenko, “Boolean technology mapping based on logic decomposition,” in Integrated Circuits and Systems Design, 2003. SBCCI 2003. Proceedings. 16th Symposium on, Sep. 2003, pp. 35–40.

[6] A. Abdollahi and M. Pedram, “Symmetry detection and boolean matching utilizing a signature-based canonical form of boolean functions,” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 27, no. 6, pp. 1128–1137, 2008.

[7] J. Mohrke and S. Malik, “Permutation and phase independent boolean comparison,” Integration the Vlsi Journal, vol. 16, no. 2, pp. 109–129, 1993.

[8] G. Micheli, Synthesis and optimization of digital circuits. New York: McGraw-Hill Higher Education, 1994.

[9] U. Hinsberger and R. Kolla, “Boolean matching for large libraries,” in Design Automation Conference, 1998. Proceedings, San Francisco, CA, USA, 1998, pp. 206–211.

[10] D. Chai and A. Kuehlmann, “Building a better boolean matcher and symmetry detector,” in Design, Automation and Test in Europe, 2006. DATE ’06. Proceedings, Munich, 2006, pp. 1–6.

[11] D. Debnath and T. Sasao, “Efficient computation of canonical form for boolean matching in large libraries,” in Design Automation Conference, 2004, Proceedings of the ASP-DAC 2004. Asia and South Pacific, Yokohama, Japan, Jan. 2004, pp. 591–596.

[12] J. Ciric and C. Sechen, “Efficient canonical form for boolean matching of complex functions in large libraries,” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 22, no. 5, pp. 535–544, 2003.

[13] Z. Huang, L. Wang, and Y. Naskovskiy, “Fast boolean matching based on npn classification,” in Field-Programmable Technology (FPT), 2013 International Conference on, Kyoto, Japan, Dec. 2013, pp. 310–313.

[14] A. Petkovska, M. Soeken, G. D. Micheli, P. Ienne, and A. Mishchenko, “Fast hierarchical npn classification,” in International Conference on Field Programmable Logic and Applications, Lausanne, Switzerland, Dec. 2016, pp. 1–4.

[15] A. Abdollahi and M. Pedram, “A new canonical form for fast boolean matching in logic synthesis and verification,” in Design Automation Conference, 2005. Proceedings. 42nd, 2005, pp. 379–384.

[16] G. Agosta, F. Bruschi, G. Pelosi, and D. Sciuto, “A unified approach to canonical form-based boolean matching,” in Design Automation Conference, 2007. DAC ’07. 44th ACM/IEEE, San Diego, CA, 2007, pp. 841–846.

[17] Y. T. Lai, S. Sastry, and M. Pedram, “Boolean matching using binary decision diagrams with applications to logic synthesis and verification,” in Computer Design: VLSI in Computers and Processors, 1992. ICCD ’92. Proceedings, IEEE 1992 International Conference on, Cambridge, MA, Oct. 1992, pp. 452–458.

[18] K.-H. Wang, T. Hwang, and C. Chen, “Exploiting communication complexity for boolean matching,” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 15, no. 10, pp. 1249–1256, 1996.

[19] L. Benini and G. Micheli, “A survey of boolean matching techniques for library binding,” ACM Transactions on Design Automation of Electronic Systems, vol. 2, no. 3, pp. 193–226, 2003.

[20] S. Chatterjee, A. Mishchenko, R. K. Brayton, X. Wang, and T. Kam, “Reducing structural bias in technology mapping,” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 25, no. 12, pp. 519–526, 2005.

[21] C. Yu, L. Wang, C. Zhang, and Y. Hu, “Fast filter-based boolean matchers,” IEEE Embedded Systems Letters, vol. 5, no. 4, pp. 65–68, 2013.

[22] Y. Hu, V. Shih, R. Majumdar, and L. He, “Exploiting symmetry in satis-based boolean matching for heterogeneous fpga technology mapping,” in Computer-Aided Design, 2007. ICCAD 2007. IEEE/ACM International Conference on, San Jose, CA, Nov. 2007, pp. 1092–3152.

[23] Y. Matsunaga, “Accelerating satis-based boolean matching for heterogeneous fpgas using one-hot encoding and cegar technique,” in Design Automation Conference (ASP-DAC), 2015 20th Asia and South Pacific, Chiba, Jan. 2015, pp. 255–260.

[24] J. Cong and Y.-Y. Hwang, “Boolean matching for lut-based logic blocks with applications to architecture evaluation and technology mapping,” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 20, no. 9, pp. 1077–1090, 2001.

[25] M. Soeken, A. Mishchenko, A. Petkovska, B. Sterin, P. Ienne, R. K. Brayton, and G. D. Micheli, “Heuristic npn classification for large functions using aigs and lexsat,” SAT, pp. 212–227, 2016.

[26] Shadi Hadi and L. Igor, “Large-scale boolean matching,” Advanced Techniques in Logic Synthesis Optimizations Applications, pp. 771–776, 2010.

[27] H. Katebi, K. A. Sakallah, and I. L. Markov, “Generalized boolean symmetries through nested partition refinement,” in IEEE/ACM International Conference on Computer-Aided Design, San Jose, CA, USA, Nov. 2013, pp. 763–770.

[28] C.-F. Lai, J.-H. R. Jiang, and K.-H. Wang, “Boolean-matching of function vectors with strengthened learning,” in Computer-Aided Design
[29] J. S. Zhang, M. Chrzanowska-Jeske, A. Mishchenko, and J. R. Burch, “Linear cofactor relationships in boolean functions,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 6, pp. 1011–1023, 2006.

Juling Zhang is currently pursuing a Ph.D. at the School of Computer Science and Engineering, University of Electronic Science and Technology of China, China. Her research interests include logic synthesis, Boolean matching and information security risk assessments.

Guowu Yang received his B.S. degree from the University of Science and Technology of China in 1989, his M.S. degree from Wuhan University of Technology in 1994, and his Ph.D. degree in electrical and computer engineering from Portland State University in 2005. He worked at the Wuhan University of Technology from 1989 to 2001 and at Portland State University from 2005 to 2006. He is currently a full professor at University of Electronic Science and Technology of China. His research interests include verification, logic synthesis, quantum computing and machine learning. He has published over 100 journal and conference papers.

William N. N. Hung received his B.S. degree and M.S. degree from the University of Texas at Austin in 1994 and 1997, respectively, and his Ph.D. from Portland State University in 2002—all in electrical and computer engineering.

He is currently a Principal Engineer at Synopsys in Mountain View, California, leading technological innovation efforts on constraint-based verification and hardware-accelerated verification, such as emulation and prototyping. He has worked at several high-tech companies, including Intel, Syncplicity and Synopsys. He has over 19 years of industrial R&D experience, has published over 80 journal and conference papers, and has patented numerous inventions.

Dr. Hung is currently an Associate Editor for IEEE Transactions on CAD and IEEE Transactions on Circuits and Systems II. He has served on the technical program committees of conferences such as DAC, DATE, ICCD, CAV, FMCAD, CEC, WCC, and others. He was the Chair of the Quantum Computing Task Force under the Emergent Technologies Technical Committee of the IEEE Computational Intelligence Society. He also served as Co-Chair of the Logic and Circuit Track for the technical program committee of ICCD.

Yan Zhang is currently pursuing a Ph.D. at the School of Computer Science and Engineering, University of Electronic Science and Technology of China, China. Her research interests include logic synthesis, optimization problems and machine learning.