# Fast Adjustable NPN Classification Using Generalized Symmetries

XUEGONG ZHOU and LINGLI WANG, State Key Lab of ASIC and System, Fudan University
ALAN MISHCHENKO, Department of EECS, UC Berkeley

NPN classification of Boolean functions is a powerful technique used in many logic synthesis and technology mapping tools in both standard cell and FPGA design flows. Computing the canonical form is the most common approach of Boolean function classification. This article proposes two different hybrid NPN canonical forms and a new algorithm to compute them. By exploiting symmetries under different phase assignment as well as higher-order symmetries, the search space of NPN canonical form computation is pruned and the runtime is dramatically reduced. Nevertheless, the runtime for some difficult functions remains high. Fast heuristic method can be used for such functions to compute semi-canonical forms in a reasonable time. The proposed algorithm can be adjusted to be a slow exact algorithm or a fast heuristic algorithm with lower quality. For exact NPN classification, the proposed algorithm is 40× faster than state-of-the-art. For heuristic classification, the proposed algorithm has similar performance as state-of-the-art with a possibility to trade runtime for quality.

CCS Concepts: • **Hardware** → *Logic synthesis*; • **Computer systems organization** → *Reconfigurable computing*;

Additional Key Words and Phrases: NPN classification, Boolean matching, symmetry, canonical form, Boolean signatures

**ACM Reference format:**
Xuegong Zhou, Lingli Wang, and Alan Mishchenko. 2019. Fast Adjustable NPN Classification Using Generalized Symmetries. *ACM Trans. Reconfigurable Technol. Syst.* 12, 2, Article 7 (April 2019), 16 pages.
http://dx.doi.org/10.1145/3313917

## 1 INTRODUCTION

Classification of Boolean functions is the task of grouping similar functions into equivalence classes. A related problem is Boolean matching, which checks whether two functions belong to the same equivalent class.

The most frequently used classification method is based on Negation-Permutation-Negation (NPN) equivalence. Two single output Boolean functions are NPN equivalent, if one can be obtained from the other by negating inputs, permuting inputs, or negating the output.

NPN classification algorithms have many applications in logic synthesis [1–4] and technology mapping [5–7] for standard cells and FPGAs. In synthesis, optimal or near-optimal circuits for a

This work was supported in part by SRC Contracts 2710.001 and 2867.001.
Authors' addresses: X. Zhou and L. Wang, State Key Laboratory of ASIC and System, Fudan University, Shanghai 201203, China; emails: {xgzhou, llwang}@fudan.edu.cn; A. Mishchenko, Electrical Engineering and Computer Sciences Department, University of California at Berkeley, Berkeley, CA 94720 USA; email: alanmi@berkeley.edu.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2019 Association for Computing Machinery.
1936-7406/2019/04-ART7 $15.00
http://dx.doi.org/10.1145/3313917

large number of practical functions can be precomputed and stored in a library. With Boolean function classification, only one function in each equivalence class needs to be in the library, resulting in a dramatic reduction of the library size [2, 4].

In technology mapping, NPN classes of functions that can be matched against a programmable cell are pre-computed and stored in a hash table to allow for a quick constant-time check whether a function is realizable using the given cell. Several recent studies are based on this approach [8, 9].

In these applications, the speed of NPN classification determines the speed of the synthesis engine or the technology mapper. This is because library precomputation can be done offline, while NPN classification is done online during runtime. Similarly, if the quality of NPN classification is poor, then more precomputation has to be done and the resulting library takes more memory.

A common approach of NPN classification is to construct a canonical form for a Boolean function *f*, and use this canonical form as the representative of the equivalence class *f* belongs to. There are several NPN canonical forms, such as the function with the smallest truth-table representation [10, 11], or with the smallest spectrum representation in the NPN equivalence class [12–14].

Naïve computation of the canonical form requires exhaustive enumeration of all the possible transformations. For NPN classification of $n$ input Boolean functions, $2^{n+1}n!$ transformations should be enumerated. Various methods based on signatures [10, 11] and variable symmetry [10, 11, 13–16] are used to reduce the computation cost. However, none of the methods works well for all functions. In some cases, a semi-canonical form can be used instead of the exact canonical form [15, 17] to get practical runtime.

The main contributions of this article are:

- An in-depth study of NPN classification as a theoretical guidance to define new signature-based canonical forms, and to determine how the canonical form computation can be adapted for a given algorithm.
- Two different hybrid NPN canonical forms introduced by combining the truth table with different Boolean signatures. One of them is suitable for heuristic classification, and the other is suitable for exact classification.
- A new classification algorithm computes the preceding canonical forms efficiently for most Boolean functions by exploiting various symmetries of the functions.
- A heuristic method is introduced to classify rare difficult functions. By adjusting the exact-to-heuristic ratio, the proposed algorithm trades runtime for the classification quality.

The proposed approach is the most general among the existing ones while also being the most practical. The proposed implementation in ABC [35] outperforms other methods in each category. In particular, the exact method is 40× faster than the best-known exact method, while the heuristic method has similar runtime but better quality and allows for a flexible quality/runtime tradeoff.

Compared to our previous work [18], this article presents a clear definition of higher-order symmetry, which is consistent with the first-order symmetry definition and simplifies the implementation. A new hybrid canonical form is proposed, which is more efficient for difficult functions, and therefore brings significant speedup to exact classification.

The remainder of this article is organized as follows: Section 2 reviews past work on NPN classification. Section 3 formally introduces the terminology. Section 4 defines the generalized variable symmetry. Section 5 defines the hybrid canonical forms. Section 6 describes the algorithm. Section 7 shows the results of the experimental evaluation, and Section 8 concludes the article.

## 2 RELATED WORK

The problems of Boolean function classification and Boolean matching have been studied since the1950s [12, 19]. Benini and Micheli [20] provide a survey of the main approaches and break them

down into three groups: canonical-form-based methods, signature-based methods, and spectral methods.

Canonical form is useful for both Boolean function classification and Boolean matching. Hinsberger et al. [21] introduce a canonical form as the function with the largest truth table in the NPN equivalence class and present a tree search method to compute the canonical form. Many improvements are proposed to efficiently compute the canonical form [11, 22].

Naïve canonical form computation and pairwise Boolean matching require exhaustive enumeration of all the possible transformations. Various Boolean signatures can reduce the enumeration effort [11, 15, 20]. Boolean signature is a compact representation of a Boolean function that characterizes intrinsic properties of the function. Normally the signature is not a complete representation of the function, but it includes information needed to distinguish nonequivalent functions. Boolean signatures are used as filters in Boolean matching to filter out the unmatched functions in the cell library [23]. Some types of Boolean signatures can distinguish input variables of a function, which is useful to compute the canonical form [11].

Spectra are special signatures based on matrix transformations, such as Hadamard [20], Walsh [12, 24], or Autocorrelation [25] transforms. The spectrum is a complete representation of a function, which can define a canonical form. Spectra are not efficient to use due to their exponential size. However, a partial spectrum can be used as a compact signature. Abdollahi and Pedram [13] define a canonical form using the satisfy count of the cofactors and propose a recursive algorithm to compute the defined canonical form. Agosta et al. [14] generalize Abdollahi's algorithm for different spectra, which shows that the satisfy count of the cofactors can be regarded as a spectrum.

Boolean function classification and Boolean matching algorithms based on Boolean satisfiability (SAT) technique have been proposed in recent years [26–28]. SAT-based methods can manipulate Boolean functions with hundreds or even thousands of input variables, but they are quite slow.

Symmetric relationship of variables is used in many Boolean function classifications and Boolean matching algorithms [11, 13, 15, 16]. Most of the existing methods only use classical symmetry without considering phase assignment. Kravets and Sakallah [31] group symmetric variables into a hierarchical structure. Symmetric relationship can be used to combine groups of variables to form higher-order symmetries. Huang et al. [15] use higher-order symmetric in their algorithm.

Yang et al. [2] classify millions of Boolean functions to create a library to store the synthesized circuit structures. Exact NPN classification is too costly for this purpose. Hence, they proposed a heuristic classification algorithm based on the three rules described in Section 5.2, which is a simplified version of Abdollahi's algorithm in Reference [13]. Huang et al. [15] have improved the classification algorithm by truth-table permutation and higher-order symmetry detection. Petkovska et al. [17] have further sped up the algorithm by reusing the intermediate results and presented a fast exact classification algorithm. These algorithms use truth tables to classify functions with up to 16 inputs. The original algorithm by Yang et al. [2] and both of its improvements are implemented in ABC [35] as command *testnpn*.

This article is an extension and generalization of the preceding classification works. It offers an in-depth review and correction of the previous methods and proposes a more efficient classification algorithm.

## 3 PRELIMINARIES

### 3.1 Boolean Function

This article deals with completely specified Boolean functions, $f(X) : B^n \rightarrow B, B = \{0, 1\}$, where $X = (x_1,\ x_2, \ldots x_n)$ is a bit vector of size $n$, $x_i \in B$. When the input bit vector is considered as a binary number, whose value is $m$, the corresponding bit vector is denoted as $X_{(m)}$. The *truth*

*table* of function $f$ is a bit vector of size $2^n$, composed of the output value of the function: $T(f) = (f(X_{(2^n-1)}), \ldots, f(X_{(1)}),\ f(X_{(0)}))$.

A *literal* $\dot{x}$ is a variable $x$ or its complement $\bar{x}$. A *cube* is the Boolean conjunction of literals. A *minterm* is a cube with $n$ literals. The *satisfy count* of a function, denoted as $|f|$, is the number of on-set minterms covered by $f$.

The *cofactor* of $f$ with respect to a literal $\dot{x}$, denoted as $f_{\dot{x}}$, is the function obtained by setting $\dot{x}$ to 1 in $f$. The cofactor of $f$ with respect to cube $c$, denoted as $f_c$, is the function obtained by setting all literals of the cube to 1.

A Boolean function $f$ is called *balanced* if $|f| = |\bar{f}|$. An input variable $x$ is called *balanced* if $|f_x| = |f_{\bar{x}}|$.

### 3.2 NPN Equivalence

*Definition 1.* An *NPN transformation* on a Boolean function is a phase assignment, followed by a permutation of its input variables, followed by a polarity assignment of its output. Applying transformation $\tau$ to function $f$ is denoted as $f \circ \tau$.

For a Boolean function of $n$ inputs, there are $2^{n+1}n!$ distinct NPN transformations. We denote a transformation $\tau$ by a vector of literals to indicate the permutation and phases of the inputs and a $z$ indicating the polarity of the output. For example, applying transformation $\tau = (\overline{x_2},\ x_1, x_3,\ \bar{z})$ to function $f(x_1,\ x_2, x_3) = x_1x_2 + x_2x_3$ results in a new function $f \circ \tau = \overline{\overline{x_2}x_1 + x_1x_3}$, which replaces $x_1$ with $\overline{x_2}$, and $x_2$ with $x_1$, respectively, while also negating the output.

*Definition 2.* Two Boolean functions, $f$ and $g$, are *NPN equivalent*, denoted as $f \equiv g$, if there exists an NPN transformation $\tau$, such that $f \circ \tau$ is equal to $g$.

The NPN equivalence is an equivalence relation, which partitions all single-output Boolean functions into equivalence classes. The *NPN equivalence class* of a function $f$ is denoted $[f]$ and is defined as the set of functions that are NPN equivalent to $f$, i.e., $[f] = \{g \mid f \equiv g\}$.

As an example, for function $f(x_1,\ x_2, x_3) = x_1x_2 + x_3$, its NPN equivalence class contains 48 functions, such as $x_1\overline{x_3} + x_2$ and $\overline{x_2x_3 + \overline{x_1}}$, with their corresponding NPN transformations $(x_1, \overline{x_3}, x_2, z)$ and $(x_2, x_3, \overline{x_1}, \bar{z})$. There are in total $2^{3+1}3!$ =96 different NPN transformations of 3 variable functions. Some transformations may produce the same function because of the variable symmetry; hence, the equivalence class contains much less functions than 96.

## 4 SYMMETRY RELATIONSHIP

### 4.1 First-Order Symmetries

*Definition 3 (Variable Symmetry).* Two variables $x_i$ and $x_j$ are said to be symmetric in function $f$, which is denoted by $x_i \leftrightarrow x_j$, if $f$ is invariant under an exchange of $x_i$ and $x_j$, i.e., $f(\ldots, x_i, \ldots,\ x_j, \ldots) = f(\ldots, x_j, \ldots,\ x_i, \ldots)$. This classical symmetry is called *nonequivalent-symmetry (NE-symmetry).* When considering the phase assignment, if $f$ is invariant under an exchange of $x_i$ and $\bar{x}_j$, i.e., $f(\ldots, x_i, \ldots,\ \bar{x}_j, \ldots) = f(\ldots, \bar{x}_j, \ldots,\ x_i, \ldots)$, then variables $x_i$ and $x_j$ are said to be *equivalent-symmetric (E-symmetric)*, which is denoted by $x_i \leftrightarrow \bar{x}_j$. If $x_i$ and $x_j$ are simultaneously NE- and E-symmetric, then $x_i$ and $x_j$ are said to be *multiform symmetric*, which is denoted by $x_i \overset{\mathrm{m}}{\leftrightarrow} x_j$.

For example, in function $f(x_1, x_2, x_3, x_4, x_5, x_6) = (x_1x_2 + x_3\bar{x}_4)(x_5 \oplus x_6)$, we have $x_1 \leftrightarrow x_2$, $x_3 \leftrightarrow \bar{x}_4$ and $x_5 \overset{\mathrm{m}}{\leftrightarrow} x_6$.

It can be shown using Boole's expansion theorem that variable symmetry is equivalent to the equality of the cofactor pair, i.e., $x_i \leftrightarrow x_j$ if and only if $f_{\bar{x}_ix_j} = f_{x_i\bar{x}_j}$; $x_i \leftrightarrow \bar{x}_j$ if and only if $f_{x_ix_j} = f_{\bar{x}_i\bar{x}_j}$ [29]. It's worth noting that the "equivalent" in the definition refers to the equivalent phase

of the literals in the cofactor, but not the equivalent phase of the two symmetric variables. There are a number of efficient algorithms in the literature for symmetry detection [29, 30].

By negating one symmetric variable of the function $f$, the E-symmetry is converted to NE-symmetry. Therefore, in the classification process, only NE-symmetry and multiform symmetry are considered. The multiform symmetry cannot be simply regarded as NE-symmetry and must be manipulated separately. Many existing classification methods [13, 15] neglected this difference.

The NE-symmetry and the multiform symmetry are both equivalence relations, while the E-symmetry is not. Hence, these equivalence relations can be used to partition the input variables $x_1,\ x_2, \ldots x_n$ into equivalence classes.

*Definition 4 (Symmetric Class).* When the variables in an equivalence class are NE-symmetric, this class is called an *NE-symmetric class*, which is denoted by $[x_{i_1}, x_{i_2}, \ldots x_{i_m}]$. When the variables in an equivalence class are multiform symmetric, this class is called a *multiform symmetric class*, which is denoted by $x_{i_1}, x_{i_2}, \ldots x_{i_m}$.

The phases of variables in NE-symmetric classes are determined, while the phases of variables in a multiform class are undetermined; an arbitrary phase can be chosen for these variables.

The following theorem is useful for manipulating the phase assignment of the multiform symmetric classes:

Lemma 1. *Let $x_i$ and $x_j$ be E-symmetric, $x_i \leftrightarrow \bar{x}_j$, then $f(\ldots, x_i, \ldots, x_j, \ldots) = f(\ldots, \bar{x}_j, \ldots,\ \bar{x}_i, \ldots)$.*

Proof. $f(\ldots, x_i, \ldots, x_j, \ldots) = f(\ldots, x_i, \ldots, \overline{\overline{x_j}}, \ldots) = f(\ldots, \bar{x}_j, \ldots,\ \bar{x}_i, \ldots)$. □

Theorem 1. *Let $x_i$ and $x_j$ be multiform symmetric, $x_i \overset{\text{m}}{\leftrightarrow} x_j$, then $f(\ldots, x_i, \ldots, x_j, \ldots) = f(\ldots, \bar{x}_i, \ldots,\ \bar{x}_j, \ldots)$.*

Proof. $f(\ldots, x_i, \ldots, x_j, \ldots) = f(\ldots, x_j, \ldots, x_i, \ldots) = f(\ldots, \bar{x}_i, \ldots,\ \bar{x}_j, \ldots)$. □

Theorem 1 indicates that, when negating an even number of the variables in a multiform symmetric class, the function is invariant. Only two different phase assignments of an $m$ variable multiform symmetric class need to be considered, instead of the whole $2^m$ phase assignments.

*Definition 5 (Negation of Symmetric Class).* If $C = [x_{i_1}, x_{i_2}, \ldots x_{i_m}]$ is an NE-symmetric class, then the negation of $C$ is the negation of each variable in $C$, i.e., $\bar{C} = [\bar{x}_{i_1}, \bar{x}_{i_2}, \ldots \bar{x}_{i_m}]$. If $C = x_{i_1}, x_{i_2}, \ldots x_{i_m}$ is a multiform symmetric class, then the negation of $C$ is the negation of one single variable in $C$, i.e., $\bar{C} = \bar{x}_{i_1}, x_{i_2}, \ldots x_{i_m}$.

With this definition, a symmetric class can be permuted and negated like a single variable. This is the basis of defining higher-order symmetries.

## 4.2 Higher-Order Symmetries

The symmetric relation of two variables can be extended to two symmetric classes as a second-order symmetry [31].

*Definition 6 (Second-order Symmetry).* For two symmetric classes $C_i = (x_{i_1}, x_{i_2}, \ldots x_{i_m})$ and $C_j = (x_{j_1}, x_{j_2}, \ldots x_{j_m})$ with the same size. If $f$ is invariant under an exchange of $C_i$ and $C_j$, i.e., $f(\ldots, x_{i_1}, \ldots, x_{i_2}, \ldots x_{i_m}, \ldots, x_{j_1}, \ldots, x_{j_2}, \ldots x_{j_m}, \ldots) = f(\ldots, x_{j_1}, \ldots, x_{j_2}, \ldots x_{j_m}, \ldots, x_{i_1}, \ldots, x_{i_2}, \ldots x_{i_m}, \ldots)$, then $C_i$ and $C_j$ are said to be NE-symmetric, which is denoted by $C_i \leftrightarrow C_j$. If $C_i$ and $C_j$ have the same type, and $f$ is invariant under an exchange of $C_i$ and $\bar{C}_j$, i.e., $f(\ldots, x_{i_1}, \ldots, x_{i_2}, \ldots x_{i_m}, \ldots, \bar{x}_{j_1}, \ldots, \bar{x}_{j_2}, \ldots \bar{x}_{j_m},\ \ldots) = f(\ldots, \bar{x}_{j_1}, \ldots, \bar{x}_{j_2}, \ldots \bar{x}_{j_m}, \ldots, x_{i_1}, \ldots, x_{i_2}, \ldots x_{i_m},\ \ldots)$ when $C_i$ and $C_j$ are both NE-symmetric classes, or

$f(\ldots, x_{i_1}, \ldots, x_{i_2}, \ldots x_{i_m}, \ldots, \bar{x}_{j_1}, \ldots, x_{j_2}, \ldots x_{j_m}, \ldots) = f(\ldots, \bar{x}_{j_1}, \ldots, x_{j_2}, \ldots x_{j_m}, \ldots, x_{i_1}, \ldots, x_{i_2}, \ldots x_{i_m}, \ldots)$ when $C_i$ and $C_j$ are both multiform symmetric classes, then $C_i$ and $C_j$ are said to be E-symmetric, which is denoted by $C_i \leftrightarrow \bar{C}_j$. If $C_i$ and $C_j$ are simultaneously NE- and E-symmetric, then $C_i$ and $C_j$ are said to be *multiform symmetric*, which is denoted by $C_i \overset{\mathrm{m}}{\leftrightarrow} C_j$.

For example, for function $f_1(x_1, x_2, x_3, x_4) = x_1x_2 + x_3x_4$, two first-order NE-symmetric classes are NE-symmetric, $[x_1, x_2] \leftrightarrow [x_3, x_4]$. For function $f_2(x_1, x_2, x_3, x_4) = (x_1 \oplus x_2)(\overline{x_3 \oplus x_4})$, two first-order multiform symmetric classes are E-symmetric, $\langle x_1, x_2\rangle \leftrightarrow \overline{\langle x_3, x_4\rangle}$. For function $f_3(x_1, x_2, x_3, x_4, x_5, x_6) = (x_1x_2 + x_2x_3 + x_1x_3)(\bar{x}_4\bar{x}_5 + \bar{x}_5\bar{x}_6 + \bar{x}_4\bar{x}_6) + (\bar{x}_1\bar{x}_2 + \bar{x}_2\bar{x}_3 + \bar{x}_1\bar{x}_3)(x_4x_5 + x_5x_6 + x_4x_6)$, two first-order NE-symmetric classes are multiform symmetric, $[x_1, x_2, x_3] \overset{\mathrm{m}}{\leftrightarrow} [x_4, x_5, x_6]$.

The second-order NE-symmetry and the second-order multiform symmetry are equivalence relations, while the second-order E-symmetry is not. By negating one symmetric class in the second-order E-symmetry, the second-order E-symmetry is converted to a second-order NE-symmetry. As a result, in the classification process, only NE-symmetry and multiform symmetry are considered. They are both equivalence relations and can partition the first-order symmetric class into second-order symmetric classes. Several first-order symmetric classes can be merged into a second-order symmetric class. The merge process can be operated recursively to generate higher-order symmetric classes.

For example, given a function $f_4(x_1, x_2, x_3, x_4, x_5, x_6) = (x_1x_2 + x_3\bar{x}_4)(x_5 \oplus x_6)$, we have $x_1 \leftrightarrow x_2$, $x_3 \leftrightarrow \bar{x}_4$ and $x_5 \overset{\mathrm{m}}{\leftrightarrow} x_6$. The input variables can be divided into three symmetric classes: $C_1 = [x_1, x_2]$, $C_2 = [x_3, \bar{x}_4]$, and $C_3 = x_5, x_6$. Because $C_1 \leftrightarrow C_2$, these two classes can be merged into a second-order symmetry class: $C_{12} = [[x_1, x_2], [x_3, \bar{x}_4]]$.

The definition of negation of symmetric class can be extended for higher-order symmetric classes. If $C_{\mathrm{H}} = [C_{i_1}, C_{i_2}, \ldots C_{i_m}]$ is a higher-order NE-symmetric class, then the negation of $C_{\mathrm{H}}$ is the negation of each lower-order class in $C_{\mathrm{H}}$, i.e., $\bar{C}_{\mathrm{H}} = [\bar{C}_{i_1}, \bar{C}_{i_2}, \ldots \bar{C}_{i_m}]$. If $C_{\mathrm{H}} = \langle C_{i_1}, C_{i_2}, \ldots C_{i_m}\rangle$ is a higher-order multiform symmetric class, then the negation of $C_{\mathrm{H}}$ is the negation of one single lower-order class in $C_{\mathrm{H}}$, i.e., $\bar{C}_{\mathrm{H}} = \langle \bar{C}_{i_1}, C_{i_2}, \ldots C_{i_m}\rangle$. The negation process can be operated recursively down to input variables.

For example, $\overline{[[x_1, x_2], [x_3, x_4]]} = [\overline{[x_1, x_2]}, \overline{[x_3, x_4]}] = [[\bar{x}_1, \bar{x}_2], [\bar{x}_3, \bar{x}_4]]$; $\overline{[\langle x_1, x_2\rangle, \langle x_3, x_4\rangle]} = [\overline{\langle x_1, x_2\rangle}, \overline{\langle x_3, x_4\rangle}] = [\langle \bar{x}_1, x_2\rangle, \langle \bar{x}_3, x_4\rangle]$; $\overline{\langle [x_1, x_2, x_3], [x_4, x_5, x_6]\rangle} = \langle \overline{[x_1, x_2, x_3]}, [x_4, x_5, x_6]\rangle = \langle [\bar{x}_1, \bar{x}_2, \bar{x}_3], [x_4, x_5, x_6]\rangle$.

# 5 CANONICAL FORM

## 5.1 Signature-Based Canonical Form

The canonical form of a function $f$ is a representative selected among functions of its NPN equivalence class $[f]$ based on some criterion. There are several different canonical forms defined in References [10, 13, 14, 32]. A more general definition follows:

*Definition 7.* Let $F_n$ be the set of Boolean functions with $n$ input variables. An *NPN canonical form* is a function that can be applied to Boolean functions and satisfies two conditions:

$$\kappa(f): F_n \rightarrow F_n, (1)\ \kappa(f) \in [f], (2)\ \forall g \in [f]: \kappa(g) = \kappa(f).$$

A Boolean signature of a function can be a scalar value or a vector of scalar value, which can be compared to each other. Furthermore, if the signature uniquely and completely specifies a function, then it defines an order of Boolean functions. A concrete canonical form can be defined by this order.

Let the signature $s$ be a one-to-one relationship between $F_n$ and a strict totally ordered co-domain, such as a subset of integer or vector of integers. A strict total order can be defined on $F_n$ based on $s$: $\forall f, g \in F_n, f <_s g$ iff $s(f) < s(g)$. Any subset $S$ of $F_n$ is strictly totally ordered, so it has a unique minimum element, denoted as $\min_s(S)$. Therefore, the canonical form based on signature $s$ can be defined as follows:

*Definition 8 (Signature-based Canonical Form).* Given a comparable signature $s$ that uniquely and completely specifies a function, the canonical form of $f$ based on $s$ is the minimum element of $[f]$ based on the order defined by $s$, i.e., $\min_s([f])$.

Because $\min_s([f])$ is a unique function in $[f]$, it conforms to the two conditions of the canonical form in Definition 7.

Various signatures are used for computing a canonical form, such as the truth table [10, 11], the satisfy count of cofactors [13, 14], spectral coefficients [12, 14], and specialized binary cost [32]. We use several signatures together to define hybrid canonical forms.

### 5.2 Cofactor Signature

The cofactor signature is composed of the satisfy count of the function and the satisfy counts of the cofactors. It is a well-known signature of Boolean functions, which is adopted in several classification algorithms [2, 11, 13, 15]. Reference [13] defines an NPN canonical form based on the cofactor signature. Several simple rules have been proposed to compute the NPN canonical form using cofactor signature [2, 13, 15].

*Rule 1 (The Output Polarity).* For a Boolean function $f$ with $n$ input variables, the polarity of the canonical form is determined by the satisfy count of $f$. If $|f| < |\bar{f}|$, or equally $|f| < 2^{n-1}$, then the polarity is positive; if $|f| > |\bar{f}|$, or equally $|f| > 2^{n-1}$, then the polarity is negative; if $f$ is balanced, then the polarity is not determined.

*Rule 2 (The Phase of Input Variables).* The phase of each input variable in the canonical form is determined by the satisfy count of the cofactor with respect to that variable. If $|f_{x_i}| < |f_{\overline{x_i}}|$, or equally $|f_{x_i}| < |f|/2$, then the phase of $x_i$ is positive; if $|f_{x_i}| > |f_{\overline{x_i}}|$, or equally $|f_{x_i}| > |f|/2$, then the phase of $x_i$ is negative; if $x_i$ is balanced, then the phase is not determined.

*Rule 3 (The Ordering of Input Variables).* After the phases of input variables are assigned, their order is determined by the ascending order of the satisfy count of the cofactors with respect to each variable. Several truth-table-based NPN canonical form algorithms use these rules [2, 15]. However, these rules may not obtain the canonical form with the minimum truth table. Instead, they produce the function with the minimum signature vector composed of the satisfy count of the function and the satisfy counts of the cofactors with respect to each variable, i.e., $(|f|, |f_{x_1}|, |f_{x_2}|, \ldots, |f_{x_n}|)$. This inspires a new canonical form combining the cofactor signatures and the truth table.

The above signature vector contains the 0th-order and the 1st-order cofactor signatures [13]. Higher-order cofactor signatures can be defined by the satisfy counts of the cofactors with respect to two or more variables, which is used in the canonical form in Abdollahi's approach [13].

### 5.3 Sum of Squared Row Sums

Row sums consist of the sums of each row of the implicant table of a function [11, 33], which is the numbers of positive variables of each minterm. For function $f(x_1, x_2, x_3, x_4) = \bar{x}_1 x_2 \bar{x}_3 x_4 + \bar{x}_1 \bar{x}_2 \bar{x}_3 + \bar{x}_1 \bar{x}_2 x_4 = \bar{x}_1 x_2 \bar{x}_3 x_4 + \bar{x}_1 \bar{x}_2 \bar{x}_3 x_4 + \bar{x}_1 \bar{x}_2 \bar{x}_3 \bar{x}_4 + \bar{x}_1 \bar{x}_2 x_3 x_4$, it has the row sums of {0,1,2,2}.

The length of row sums varies from 0 to $2^n$ for $n$ input functions. This makes it difficult to use in the classification algorithm. However, it can be reduced to a single value by adding the square of

each value together, which is called *sum of squared row sums (SSRS)* [11]. For the above example, $SSRS(f) = 0^2+1^2+2^2+2^2 = 9$.

The SSRS of a function is invariant under variable permutation, while it varies under variable negation. Therefore, it is useful to distinguish different phase assignments of input variables [11] and can be integrated into the canonical form.

### 5.4 Hybrid Canonical Form

We integrate different signatures into a truth table to form hybrid signatures, then define hybrid canonical forms based on these signatures.

*Definition 9 (Hybrid Signatures).* The *hybrid cofactor signature* of a function $f$, denoted by $H_c(f)$, is a vector composed of the satisfy count of the function and the satisfy counts of the cofactors with respect to each variable, followed by the bits of the truth table, i.e., $H_c(f) = (|f|,\ |f_{x_1}|,\ |f_{x_2}|,\ \ldots,\ |f_{x_n}|,\ f(X_{(2^n-1)}), \ldots,\ f(X_{(1)}),\ f(X_{(0)}))$. The *hybrid SSRS signature* of a function $f$, denoted by $H_s(f)$, can be obtained by Inserting SSRS of the function into $H_c(f)$, i.e., $H_s(f) = (|f|,\ |f_{x_1}|,\ |f_{x_2}|,\ \ldots,\ |f_{x_n}|, SSRS(f)\ , f(X_{(2^n-1)}), \ldots,\ f(X_{(1)}),\ f(X_{(0)}))$.

Both hybrid signatures contain a truth table. Therefore, they uniquely and completely specify a function. The signatures of different functions can be compared via the lexicographic order, which is a strict total order. According to Definition 8, NPN canonical forms can be defined based on $H_c(f)$ or $H_s(f)$.

*Definition 10 (Hybrid Canonical Forms).* For a Boolean function $f$, its *hybrid cofactor canonical form* is the function in $[f]$ with the minimum *hybrid cofactor signature*, denoted as $\kappa_c(f)$. Its *hybrid SSRS canonical form* is the function in $[f]$ with the minimum *hybrid SSRS signature*, denoted as $\kappa_s(f)$.

In the hybrid canonical form, the cofactor signature and SSRS take precedence over the truth table. They are used to determine the phase and the order of each input variable. The truth table is used to distinguish different functions. The hybrid cofactor canonical form is easier to compute and is suitable for heuristic classification, while the hybrid SSRS canonical form is suitable for exact classification.

Unlike Abdollahi's approach [13], the hybrid canonical forms contain 0th-order and 1st-order cofactor signatures only, but not higher-order signatures. Abdollahi's approach is BDD-based and therefore needs higher-order signatures to distinguish individual functions, while the hybrid canonical forms use truth tables to distinguish functions. We have tried to integrate 2nd-order cofactor signatures into the canonical form, but the complexity of manipulating them negates their benefits.

### 5.5 Variable Symmetry and Canonical Form

As described in Section 3, the input variables of a function can be partitioned into symmetric classes. The variables in a symmetric class have the same properties. During the canonical form computation, a symmetric class is handled as a single variable, resulting in the dramatic reduction of the complexity of the algorithm. This method was adopted in several algorithms [13–15], but the correctness is not proven.

Actually, this method is invalid for the spectrum-based canonical form used in References [13] and [14], because it neglects the higher-order coefficients among the variables in a symmetric class. However, the canonical form can be redefined to make this method feasible. Theorem 2, which follows, shows that the variable grouping method can be applied to all the signature-based canonical forms defined by Definition 8.

*Definition 11.* A Boolean function $f$ is *symmetry aggregate* if all of its symmetric variables are placed in adjacent positions within a group in the input variable vector.

For example, function $f_1(x_1, x_2, x_3, x_4, x_5, x_6) = x_1\bar{x}_2x_3 + x_4 + x_5x_6$ is symmetry aggregate, because variables in both symmetric classes$[x_1, x_2, x_3]$ and $[x_5, x_6]$ are placed in adjacent positions in the input variable vector; while $f_2(x_1, x_2, x_3, x_4, x_5, x_6) = x_1\bar{x}_2x_4 + x_3 + x_5x_6$ is not symmetry aggregate, because the variables in symmetric group $[x_1, x_2, x_4]$ are separated in the input variable vector.

*Definition 12.* The *symmetry aggregate equivalence set* of a Boolean function $f$, denoted by $[f]^S$, is the set of the symmetry aggregate functions in the NPN equivalence class of $f$, i.e., $[f]^S = \{h \mid h \in [f], h \text{ is symmetry aggregate}\}$.

Theorem 2. *When a strict total order is defined on $F_n$, the minimum element of $[f]^S$, denoted by $\kappa^S(f)$, is an NPN canonical form of $f$.*

Proof. Examine the two conditions of the canonical form:

(1) According to Definition 8 and Definition 12, $\min([f]^S) \in [f]^S \in [f]$.
(2) It is true that $\forall g \in [f]$ , $[f] = [g]$. According to Definition 12, $[f]^S = [g]^S$, therefore $\kappa^S(g) = \min([g]^S) = \min([f]^S)= \kappa^S(f)$ □

If the canonical form computation algorithm groups symmetric variables together, the output result of the algorithm is a *symmetry aggregate* function, which is the *symmetry aggregate* canonical form defined in Theorem 2. The proposed algorithm introduced in Section 5 deals with the symmetry aggregate hybrid canonical form defined via the hybrid signature vector.

Theorem 2 shows that the canonical form can be redefined according to the canonical form computing algorithm. This provides more freedom for the algorithm design.

## 6 COMPUTING THE CANONICAL FORM

### 6.1 Basic Canonical Form Algorithm

Given a Boolean function, the proposed adjustable algorithm computes the *symmetry aggregate hybrid cofactor canonical form* or the *symmetry aggregate hybrid SSRS canonical form* of the function. The result can be an exact canonical form or a semi-canonical form, according to certain threshold. The algorithm is organized into seven steps, as described below.

**ALGORITHM 1:** Compute the hybrid canonical form

**Input:** Boolean function $f$ with $n$ input variables, result type (cofactor or SSRS), exact threshold
**Output:** canonical form $\kappa(f)$ or semi-canonical form $\kappa'(f)$

1: Decide the polarity of the output and the phases of input variables.
2: Reorder and group the input variables by the cofactor signature.
3: Detect variable symmetry and group symmetric variables.
4: If the result type is SSRS, find the phase assignments with the minimum SSRS.
5: Estimate the cost of the exhaustive enumeration.
6: If the cost is larger than the exact threshold, do the simple enumeration, which generates $\kappa'(f)$,
7: Else do the exhaustive enumeration, which generates $\kappa(f)$.

In Step 1, the output polarity is determined according to Rule 1 described in Section 4.2. Negate the function if the output polarity is negative. Then the input phase assignment of each variable is performed according to Rule 2. Negate the variables whose phase are negative. Recalculate the cofactor signature if necessary. If $f$ is balanced, then the output polarity is undecided, and the

algorithm should use both positive and negative polarities, and the resulting function with the smaller truth table is returned.

In Step 2, based on Rule 3, the input variables are reordered such that $|f_{x_1}| \leq |f_{x_2}| \leq \cdots \leq |f_{x_n}|$. Then the variables are grouped into groups $G_1, G_2, \ldots, G_k$ by the satisfy count of the cofactors with respect to each variable. Variables with the same satisfy count are in the same group. If $f$ contains balanced variables, then all of them are grouped in the last group $G_k$. This group is called a balanced group.

In Step 3, the variables within each group are checked for symmetry and are divided into symmetric classes. If the group is balanced, then the symmetric relation can be NE-symmetry, E-symmetry, or multiform symmetry. If E-symmetry is detected, then negate one variable to convert the symmetric relation into NE-symmetry. If the group is not balanced, then only NE-symmetry needs to be checked. Then higher-order symmetries are detected, and the lower-order symmetric classes are merged into higher-order symmetric classes.

After Step 3, the input variables are grouped into several groups, and each group is divided into several symmetric classes, i.e., $G_1 = (C_1, C_2, \ldots, C_{m_1}), G_2 = (C_{m_1+1}, C_{m_1+2}, \ldots, C_{m_2}), \ldots, G_k = (C_{m_{k-1}+1}, C_{m_{k-1}+2}, \ldots, C_m)$, $1 \leq m \leq k, 1 < m_1 < m_2 < \cdots < m_{k-1} < m$. Each symmetric class $C_i$ contains one or more symmetric variables. If $G_k$ is not balanced, then the number of balanced symmetric classes $n_{\mathrm{B}} = m - m_{k-1}$, else $n_{\mathrm{B}} = 0$.

If the result type is SSRS, Step 4 enumerates phase assignments of the balanced group, calculates SSRS for each different phase assignment, and records the assignments with the minimum SSRS into a candidate assignment vector $V_{\mathrm{CA}}$. There are $2^{n_{\mathrm{B}}}$ different phase assignment needs to be enumerated.

In Step 5, three factors are used to estimate the cost of the exhaustive enumeration: the truth table manipulating cost $c_{\mathrm{t}} = n$ (the number of input variables); the permutation cost $c_{\mathrm{p}} = \log_2(m_1!(m_2 - m_1)! \ldots (m - m_{k-1})!)$; and if the result type is SSRS, the negation cost $c_{\mathrm{n}} = \log_2(\mathrm{size}(V_{\mathrm{CA}}))$, else $c_{\mathrm{n}} = n_{\mathrm{B}}$.

We use the logarithm of the exhaustive enumeration runtime as the enumeration cost and assume it is a linear combination of $c_{\mathrm{p}}$, $c_{\mathrm{n}}$, and $c_{\mathrm{t}}$. We recorded $c_{\mathrm{p}}$, $c_{\mathrm{n}}$, $c_{\mathrm{t}}$, and the runtime of each function in the experiment and computed the coefficients via linear regression.

Step 6 is a fast greedy enumeration method introduced by Huang et al. [15]. Adjacent symmetric classes in each group are swapped and flipped. A total of eight transformations are considered $(ab, a\bar{b}, \bar{a}b, \overline{ab}, ba, b\bar{a}, \bar{b}a, \overline{ba})$ if the group is balanced. Otherwise, only two transformations are considered $(ab, ba)$. The transformation leading to the smallest truth table is chosen. This procedure is repeated until there is no improvement. This step generates a semi-canonical form.

Step 7 exhaustively enumerates all the different transformations and save the minimum truth table as the canonical form. In total, $2^{c_{\mathrm{p}}}$ different permutations and $2^{c_{\mathrm{n}}}$ different phase assignments need to be enumerated. This step generates an exact canonical form.

Instead of the recursive enumeration used in many previous algorithms [13–15], we use iterative enumeration. By using the Johnson-Trotter permutation algorithm [34] and Gray code, each enumeration step only swaps one pair of adjacent symmetric classes or flips one symmetric class. Thus, the transformation cost during enumeration is minimized.

### 6.2 Hierarchical Canonical Form Algorithm

Petkovska et al. [17] introduced a hierarchical method, which reuses the intermediate results to speed up the classification process. This method can be used with the algorithm described in the previous section.

The hierarchical adjustable algorithm has three intermediate levels and maintains a hash map in each level, which maps the intermediate result to the final canonical form. This allows the algorithm to finish as soon as the intermediate result hits the hash map. The first level is after deciding the phases of variables (Step 1). The second level is after grouping symmetric variables (Step 3). The simple enumeration (Step 6) executes unconditionally, and its result is used as the key in the third-level hash map.

### 6.3 Canonical Example

Given a function $f(x_1, x_2, x_3, x_4, x_5, x_6) = (x_1x_2 + x_3\bar{x}_4)(x_5 \oplus x_6)$, we have the cofactor signature vector $(|f|, |f_{x_1}|, |f_{x_2}|, |f_{x_3}|, |f_{x_4}|, |f_{x_5}|, |f_{x_6}|) = (14, 4, 4, 4, 10, 7, 7)$. Since $|f| < 32$, the output polarity is positive.

Then perform the phase assignment. Because $|f_{x_1}| = |f_{x_2}| = |f_{x_3}| < \frac{|f|}{2}, |f_{x_4}| > \frac{|f|}{2}, |f_{x_5}| = |f_{x_6}| = \frac{|f|}{2}$, the phase of $x_1$, $x_2$, and $x_3$ is positive, the phase of $x_4$ is negative, and $x_5$ and $x_6$ are balanced. The cofactor signature vector is recalculated as $(|f|, |f_{x_1}|, |f_{x_2}|, |f_{x_3}|, |f_{\bar{x}_4}|, |f_{x_5}|, |f_{x_6}|) = (14, 4, 4, 4, 4, 7, 7)$.

In Step 2, the variables are grouped into two groups: $G_1 = (x_1, x_2, x_3, x_4)$ and $G_2 = (x_5, x_6)$, where $G_2$ is balanced.

In Step 3, the variable symmetry is detected, and the input variables are divided into three symmetric classes: $C_1 = [x_1, x_2]$, $C_2 = [x_3, x_4]$, and $C_3 = \langle x_5, x_6 \rangle$. Note that variable $x_4$ has been negated in Step 2. Then higher-order symmetry is detected and $C_1$ and $C_2$ are merged into a second-order symmetric class $C_{12} = [[x_1, x_2], [x_3, x_4]]$. After Step 3, the structure of input variables is $G_1 = ([x_1, x_2, x_3, x_4])$,$G_2 = (\langle x_5, x_6 \rangle)$.

If the result type is SSRS, Step 4 enumerates phase assignments of the balanced group $G_2$. Since it contains only one symmetric class $C_3$, two different phase assignments are enumerated. As $C_3$ is a multiform class, only one variable in $C_3$ is to be negated. The functions corresponding to the phase assignments are $f_1 = (x_1x_2 + x_3x_4)(x_5 \oplus x_6)$ and $f_2 = (x_1x_2 + x_3x_4)(\bar{x}_5 \oplus x_6)$. $SSRS(f_1) = 214$, $SSRS(f_2) = 228$. Since $SSRS(f_1) < SSRS(f_2)$, $f_1$ is added to the candidate assignment vector $V_{\mathrm{CA}}$.

Then Step 7 enumerates all the different permutations of each candidate function. Because each of the two groups contains only one symmetric class, no permutation needs to be performed. The only candidate function $f_1$ is returned as the canonical form $\kappa_{\mathrm{S}}(f)$.

If the result is the cofactor canonical form, Step 4 is skipped and Step 7 enumerates all the different permutations and all the different phase assignments. As explained above, no permutation is needed, and two different phase assignments are enumerated. The corresponding functions are $f_1 = (x_1x_2 + x_3x_4)(x_5 \oplus x_6)$ and $f_2 = (x_1x_2 + x_3x_4)(\bar{x}_5 \oplus x_6)$. The truth table $T(f_1)$ =0000 F888 F888 0000H, $T(f_2)$ =F888 0000 0000 F888H. As $T(f_1) < T(f_2)$, the canonical form of $f$ is $\kappa_{\mathrm{C}}(f) = f_1$.

## 7 EXPERIMENTAL RESULTS

Petkovska's hierarchical NPN classification algorithms [17] are the state-of-the-art algorithms, which include the hierarchical algorithm *HierH* and the exact algorithm *HierE2*. Both of them are implemented in ABC [35] (command "*testnpn -A 7*"). The proposed hierarchical adjustable algorithm is also implemented in ABC (command "*testnpn -A 8*"). We use different names to represent the proposed algorithm with different output: *AdjCH* for heuristic hybrid cofactor canonical form, *AdjCE* for exact hybrid cofactor canonical form, *AdjSH* for heuristic hybrid SSRS canonical form, and *AdjSE* for exact hybrid SSRS canonical form.

The benchmarks used in the experiments are from References [15] and [17]. They are Boolean functions divided into several test suites by their disjoint-support decomposition (DSD) properties [36] (full DSD, partial DSD, and non-DSD) and by the number of input variables. All experiments ran on a computer with a 3.1GHz Intel Core i5 CPU and 8GB main memory.

Table 1. Comparison of Heuristic Classification Algorithms

| DSD | # Vars | # Funcs | *HierH* #Classes | *HierH* Time | *AdjCH* (Number of classes / Runtime(s) / Exact ratio(%)) Threshold = 0 | | | Threshold = 25 | | | Threshold = 35 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Full | 6 | 1M | 204 | 0.06 | 215 | 0.07 | 0 | 191 | 0.07 | 100 | 191 | 0.07 | 100 |
| | 8 | 1M | 1,344 | 0.15 | 1,393 | 0.18 | 0 | 1,274 | 0.20 | 99.996 | 1,274 | 0.21 | 100 |
| | 10 | 100K | 1,723 | 0.05 | 1,729 | 0.08 | 0 | 1,713 | 0.09 | 99.7 | 1,707 | 0.14 | 99.999 |
| | 12 | 100K | 3,157 | 0.19 | 3,154 | 0.28 | 0 | 3,145 | 0.29 | 99.6 | 3,139 | 0.42 | 99.9 |
| | 14 | 10K | 891 | 0.13 | 898 | 0.24 | 0 | 893 | 0.24 | 92.9 | 883 | 0.88 | 98.8 |
| | 16 | 10K | 1,057 | 0.49 | 1,059 | 0.82 | 0 | 1,056 | 0.85 | 94.6 | 1,055 | 1.05 | 96.4 |
| Partial | 6 | 1M | 2,254 | 0.07 | 2,258 | 0.09 | 0 | 2,106 | 0.09 | 99.99 | 2,103 | 0.10 | 100 |
| | 8 | 1M | 14,270 | 0.18 | 14,268 | 0.28 | 0 | 13,944 | 0.36 | 97.9 | 13,923 | 2.25 | 99.99 |
| | 10 | 100K | 6,620 | 0.07 | 6,593 | 0.12 | 0 | 6,520 | 0.15 | 96.4 | 6,502 | 1.06 | 99.6 |
| | 12 | 100K | 8,545 | 0.23 | 8,482 | 0.40 | 0 | 8,430 | 0.50 | 86.9 | 8,413 | 1.87 | 98.4 |
| | 14 | 10K | 2,482 | 0.27 | 2,472 | 0.49 | 0 | 2,460 | 0.55 | 83.9 | 2,454 | 2.56 | 94.4 |
| | 16 | 10K | 3,110 | 1.19 | 3,101 | 2.49 | 0 | 3,089 | 2.72 | 77.4 | 3,082 | 5.99 | 89.0 |
| Non | 6 | 1M | 1,748 | 0.04 | 1,744 | 0.05 | 0 | 1,674 | 0.06 | 99.99 | 1,673 | 0.06 | 100 |
| | 8 | 1M | 2,949 | 0.09 | 2,936 | 0.10 | 0 | 2,857 | 0.15 | 98.3 | 2,836 | 0.73 | 99.9 |
| | 10 | 100K | 2,016 | 0.03 | 2,019 | 0.05 | 0 | 1,954 | 0.10 | 63.8 | 1,920 | 1.11 | 89.9 |
| | 12 | 100K | 1,409 | 0.10 | 1,393 | 0.13 | 0 | 1,364 | 0.17 | 61.8 | 1,327 | 3.21 | 91.5 |
| | 14 | 10K | 980 | 0.09 | 972 | 0.16 | 0 | 968 | 0.18 | 42.3 | 957 | 3.64 | 74.9 |
| | 16 | 10K | 282 | 0.16 | 281 | 0.24 | 0 | 281 | 0.26 | 24.6 | 281 | 0.87 | 56.9 |
| Geomean | | | | 0.13 | | 0.19 | | | 0.22 | | | 0.72 | |

### 7.1 Heuristic Classification Results

Table 1 compares the heuristic classification results of *AdjCH* with *HierH*. The two columns for *HierH* are the number of classes produced and its runtime. For *AdjCH*, the results for three different enumeration thresholds are presented. In addition to the number of classes produced and the runtime, the exact ratio is shown, which is the ratio of the exhaustively enumerated functions to the total number of functions.

A heuristic classification performs better if it generates fewer classes; that is, closer to the exact result. The pure heuristic version of *AdjCH* (Threshold = 0) has a similar classification result to *HierH*. It is slightly worse than *HierH* for full DSD functions and is slightly better than *HierH* for partial DSD and non-DSD functions. *AdjCH* is about 1.5x slower than *HierH*. Setting a proper enumeration threshold can improve the classification quality significantly with small runtime penalty. Raising the enumeration threshold increases the exact ratio further and results in better classification quality, but the runtime grows rapidly.

Table 2 shows classification results of *AdjSH*. It has better classification quality than *AdjCH*, but it is quite slow. The results indicate that computing SSRS is costly, and the hybrid SSRS canonical form is not suitable for heuristic classification.

### 7.2 Exact Classification Results

Table 3 compares the exact classification result of *AdjCE, AdjSE,* and *HierE2*. *HierE2* is not scalable enough for classifying functions with more than 10 input variables, while *AdjCE* and *AdjSE* can classify full DSD functions with 16 inputs and partial DSD functions with 12 inputs with affordable runtime. On average, *AdjCE* is 31 times faster than *HierE2*, and *AdjCE* is 41 times faster than *HierE2* when classifying functions with no more than 10 inputs.

Table 2. Classification Results of *AdjSH* Algorithm

| | # Vars | | Number of classes / Runtime(s) / Exact ratio(%) | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| DSD | | # Funcs | Threshold = 0 | | | Threshold = 25 | | | Threshold = 30 | | |
| Full | 6 | 1M | 202 | 0.23 | 0 | 191 | 0.23 | 100 | 191 | 0.23 | 100 |
| | 8 | 1M | 1,334 | 0.43 | 0 | 1,274 | 0.43 | 100 | 1,274 | 0.44 | 100 |
| | 10 | 100K | 1,720 | 0.16 | 0 | 1,707 | 0.18 | 99.998 | 1,707 | 0.27 | 100 |
| | 12 | 100K | 3,150 | 0.68 | 0 | 3,138 | 0.73 | 99.8 | 3,138 | 1.37 | 99.997 |
| | 14 | 10K | 884 | 1.99 | 0 | 882 | 2.12 | 95.7 | 882 | 3.35 | 98.8 |
| | 16 | 10K | 1,045 | 8.95 | 0 | 1,041 | 9.01 | 95.6 | 1,041 | 13.57 | 98.0 |
| Partial | 6 | 1M | 2,175 | 0.28 | 0 | 2,103 | 0.30 | 100 | 2,103 | 0.30 | 100 |
| | 8 | 1M | 14,117 | 0.91 | 0 | 13,923 | 1.32 | 99.99 | 13,923 | 1.50 | 100 |
| | 10 | 100K | 6,571 | 0.75 | 0 | 6,503 | 0.99 | 99.2 | 6,494 | 1.99 | 100 |
| | 12 | 100K | 8,484 | 4.50 | 0 | 8,424 | 4.96 | 95.5 | 8,406 | 11.35 | 99.2 |
| | 14 | 10K | 2,464 | 7.73 | 0 | 2,455 | 8.19 | 92.0 | 2,453 | 10.79 | 95.7 |
| | 16 | 10K | 3,093 | 69.56 | 0 | 3,083 | 70.19 | 84.2 | 3,074 | 80.23 | 91.2 |
| Non | 6 | 1M | 1,699 | 0.22 | 0 | 1,673 | 0.23 | 100 | 1,673 | 0.23 | 100 |
| | 8 | 1M | 2,926 | 0.34 | 0 | 2,836 | 0.46 | 99.9 | 2,836 | 0.55 | 100 |
| | 10 | 100K | 2,060 | 0.49 | 0 | 1,943 | 0.77 | 88.4 | 1,918 | 14.44 | 92.7 |
| | 12 | 100K | 1,418 | 5.38 | 0 | 1,352 | 5.88 | 82.9 | 1,321 | 17.40 | 95.5 |
| | 14 | 10K | 997 | 6.12 | 0 | 990 | 6.41 | 55.0 | 966 | 15.42 | 78.8 |
| | 16 | 10K | 283 | 15.16 | 0 | 283 | 15.23 | 39.0 | 282 | 16.41 | 59.9 |
| Geomean | | | | 1.51 | | | 1.69 | | | 2.81 | |

*AdjSE* is slower than *AdjCE* for classifying small functions and is much faster than *AdjCE* for functions with more than 8 inputs. *AdjSE* is, on average, two times faster than *AdjCE*. When considering test suites with 8 or more inputs, *AdjSE* is 7.6 times faster than *AdjCE*. Although SSRS is costly to compute, it is worthwhile to reduce the exhaustive enumeration cost in Step 7 of the algorithm.

*AdjCE* is still unable to classify the partial DSD test suite with 16 inputs and the non-DSD test suites with more than 10 inputs. *AdjSE* is unable to classify the non-DSD test suites with more than 12 inputs. The runtimes of classifying these test suites in Table 3 are estimated using the enumeration cost in Step 5 of the algorithm.

Analyzing the runtime cost of individual functions shows that, for exact classification, a few outlier functions dominate the runtime. Figure 1 shows the results of classifying the partial DSD test suite with 14 inputs. The total run time of *AdjCE* is 13.1h. Among the 10,000 functions in the test suite, 2,472 of them are processed for exhaustive enumeration, and the other functions are skipped by the hierarchical mechanism. The top 2 difficult functions cost 9.8h, and the next 10 functions cost 2.9h. The total runtime of the other 2,460 functions is only 16m.

One of the most difficult functions is $f_w = \overline{((x_{10}x_1 + \bar{x}_{10}(x_7x_2 + \bar{x}_7x_6))} + x_8)(((x_3 + x_4 + x_{13}) \oplus x_5) + \bar{x}_{12})(x_9 + x_{11}) \oplus x_{14}$. It has 11 symmetric classes, and 10 of them are balanced. Therefore, *AdjCE* needs to enumerate $2^{10}10!$ transformations, which takes 4.9h.

For *AdjSE*, the total run time for classifying the same test suite is 1,146s. The top 15 difficult functions take 1,007s, and the other 2,457 functions take only 139s. For the above difficult function $f_w$, *AdjSE* separates the phase enumeration and the permutation enumeration. Step 4 enumerates $2^{10}$ phase assignments and finds 8 candidate phase assignments. As a result, Step 7 only needs to enumerate $8 \bullet 10!$ transformations. *AdjSE* only takes 22.5s to compute $\kappa_S(f_w)$. This demonstrates how *AdjSE* reduces the enumeration cost for difficult functions.

Table 3. Classification Results of Exact Algorithms

| DSD | # Vars | # Funcs | # Classes | Runtime(s) *HierE2* | Runtime(s) *AdjCE* | Runtime(s) *AdjSE* | Speedup *AdjCE* to *HierE2* | Speedup *AdjSE* to *HierE2* | Speedup *AdjSE* to *AdjCE* |
|---|---|---|---|---|---|---|---|---|---|
| Full | 6 | 1M | 191 | 0.29 | 0.07 | 0.22 | 4.14 | 1.32 | 0.32 |
| | 8 | 1M | 1,274 | 42.9 | 0.20 | 0.42 | 214.50 | 102.14 | 0.48 |
| | 10 | 100K | 1,707 | 7,049.13 | 0.63 | 0.28 | 11,189.10 | 25,175.46 | 2.25 |
| | 12 | 100K | 3,138 | - | 1.62 | 1.44 | | | 1.13 |
| | 14 | 10K | 882 | - | 98.82 | 18.80 | | | 5.26 |
| | 16 | 10K | 1,041 | - | 550.09 | 249.50 | | | 2.20 |
| Partial | 6 | 1M | 2,103 | 0.55 | 0.10 | 0.29 | 5.50 | 1.90 | 0.34 |
| | 8 | 1M | 13,932 | 96.28 | 5.03 | 1.48 | 19.14 | 65.05 | 3.40 |
| | 10 | 100K | 6,494 | 4,662.84 | 8.58 | 1.88 | 543.45 | 2,480.23 | 4.56 |
| | 12 | 100K | 8,396 | - | 923.42 | 66.17 | | | 13.96 |
| | 14 | 10K | 2,447 | - | 13h | 1,146.29 | | | 40.98 |
| | 16 | 10K | 3,055 | - | >6,000h* | 76h | | | - |
| Non | 6 | 1M | 1,673 | 0.33 | 0.08 | 0.22 | 4.13 | 1.50 | 0.36 |
| | 8 | 1M | 2,836 | 12.66 | 1.83 | 0.54 | 6.92 | 23.44 | 3.39 |
| | 10 | 100K | 1,904 | 3,566.01 | 1,272.60 | 220.79 | 2.80 | 16.15 | 5.76 |
| | 12 | 100K | 1,304 | - | >100h* | 62h | - | - | - |
| | 14 | 10K | - | - | >100kh* | >150h* | - | - | - |
| | 16 | 10K | - | - | >500h* | >300h* | - | - | - |
| Geomean | | | | | | | 32.93 | 43.77 | 2.27 |

*estimated time.

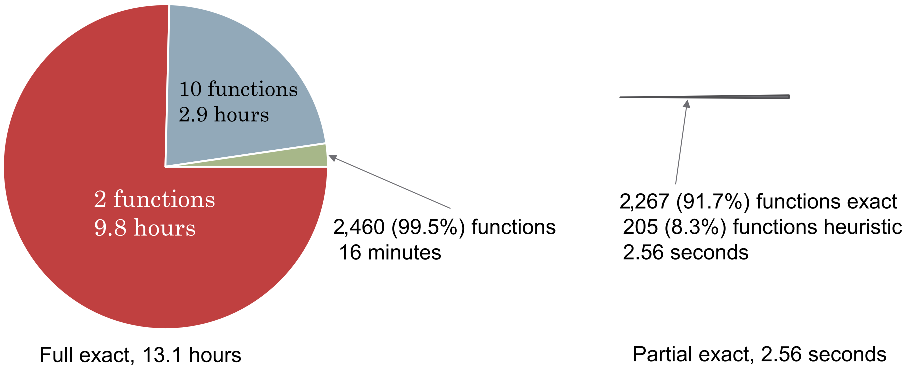


Fig. 1. Runtime cost for classifying the partial DSD test suite with 14 inputs.

Comparing the exact runtime of *AdjCE* to the heuristic runtime (threshold = 35) of *AdjCH* also shows the disproportionate runtime of a few functions, as shown in Table 4. Take the partial DSD test suite with 14 inputs as an example: The heuristic classification generates 94.4% exact canonical form and only cost 0.005% of the full exact runtime. In other words, for exact classification, 5.6% of the functions occupy more than 99.995% of the runtime. The precise result considering hierarchical mechanism is shown in Figure 1. *AdjCH* processes 2,261 functions for exhaustive enumeration, which is 91.7% of the functions processed by *AdjCE*. This is the reason why *AdjCH* performs effective heuristic classification. Most of the functions are processed exactly, and only a few difficult functions are processed by the fast heuristic method.

Table 4. Runtime Comparison of Heuristic and Exact Classification

| DSD Property | # Vars | Heuristic Time(s) | Exact Time(s) | Runtime Ratio(%) | Function Ratio(%) |
|---|---|---|---|---|---|
| Full | 10 | 0.14 | 0.20 | 70.0 | 99.999 |
| | 12 | 0.42 | 0.63 | 66.7 | 99.9 |
| | 14 | 0.88 | 1.62 | 54.3 | 98.8 |
| | 16 | 1.05 | 98.82 | 1.1 | 96.4 |
| Partial | 8 | 2.25 | 5.03 | 44.7 | 99.99 |
| | 10 | 1.06 | 8.58 | 12.4 | 99.6 |
| | 12 | 1.87 | 923.42 | 0.2 | 98.4 |
| | 14 | 2.56 | 13h | 0.005 | 94.4 |
| Non | 8 | 0.73 | 1.83 | 39.9 | 99.9 |
| | 10 | 1.11 | 1,272.60 | 0.1 | 89.9 |

## 8 CONCLUSION

This article presents an adjustable NPN classification algorithm, which can be either exact or heuristic. As a heuristic algorithm, the proposed algorithm can be adjusted to trade off the runtime and the classification quality. As an exact classification algorithm, the proposed algorithm is faster than state-of-the-art. The main reason of the speedup is that the algorithm takes full advantage of various variable symmetries, especially the proper manipulation of the multiform symmetric groups ignored by many of the existing classification methods.

By integrating more signatures into the canonical form, it can distinguish different functions better, thus reducing the enumeration cost for exact classification and gaining a significant speedup. However, in the heuristic classification, the extra computation cost reduces its benefit.

The SSRS signature is not powerful enough to deal with exact classification of non-DSD functions with more than ten inputs. In future work, we will analyze the difficult functions and find efficient signatures to handle them.

## REFERENCES

[1] A. Kennings, A. Mishchenko, K. Vorwerk, V. Pevzner, and A. Kundu. 2010. Efficient FPGA resynthesis using precomputed LUT structures. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'10)*. 532–537.

[2] W. Yang, L. Wang, and A. Mishchenko. 2012. Lazy man's logic synthesis. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'12)*. 597–604.

[3] M. Soeken, L. G. Amarù, P. Gaillardon, and G. De Micheli. 2016. Optimizing majority-inverter graphs with functional hashing. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE'16)*.

[4] A. Kennings, A. Mishchenko, K. Vorwerk, V. Pevzner, and A. Kundu. 2010. Generating efficient libraries for use in FPGA resynthesis algorithms. In *Proceedings of the International Workshop on Logic and Synthesis (IWLS'10)*. 147–154.

[5] S. Chatterjee, A. Mishchenko, R. K. Brayton, X. Wang, and T. Kam. 2006. Reducing structural bias in technology mapping. *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.* 25, 12 (2006), 2894–903.

[6] A. Mishchenko, S. Cho, S. Chatterjee, and R. Brayton. 2007. Combinational and sequential mapping with priority cuts. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'07)*. 354–361.

[7] A. Mishchenko, R. Brayton, W. Feng, and J. W. Greene. 2015. Technology mapping into general programmable cells. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'15)*. 70–73.

[8] S. Ray, A. Mishchenko, N. Een, R. Brayton, S. Jang, and C. Chen. 2012. Mapping into LUT structures. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE'12)*. 1579–1584.

[9] W. Feng, J. W. Greene, and A. Mishchenko. 2018. Improving FPGA performance with a S44 LUT structure. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'18)*. 61–66.

[10] U. Hinsberger and R. Kolla. 1998. Boolean matching for large libraries. In *Proceedings of the Design Automation Conference (DAC'98)*, 206–211.

[11] D. Chai and A. Kuehlmann. 2016. Building a better Boolean matcher and symmetry detector. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE'16).* 1079–1084.
[12] S. W. Golomb. 1959. On the classification of Boolean functions. *IRE Trans. Circuit Theory* CT-6 (1959), 176–186.
[13] A. Abdollahi and M. Pedram. 2008. Symmetry detection and Boolean matching utilizing a signature-based canonical form of Boolean functions. *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.* 27, 6 (2008), 1128–1137.
[14] G. Agosta, F. Bruschi, G. Pelosi, and D. Sciuto. 2009. A transform-parametric approach to Boolean matching. *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.* 28, 6 (2009), 805–817.
[15] Z. Huang, L. Wang, Y. Nasikovskiy, and A. Mishchenko. 2013. Fast Boolean matching based on NPN classification. In *Proceedings of the International Conference on Field Programmable Technology (ICFPT'13).* 310–313.
[16] C. C. Tsai, M. Marek-Sadowska, and D. Gatlin. 1997. Boolean function classification via fixed polarity Reed-Muller forms. *IEEE Trans. Comput.* 46, 2 (1997), 173–186.
[17] A. Petkovska, M. Soeken, G. De Micheli, P. Ienne, and A. Mishchenko. 2016. Fast hierarchical NPN classification. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'16)*, 61–64.
[18] X. Zhou, L. Wang, P. Zhao, and A. Mishchenko. 2018. Fast adjustable NPN classification using generalized symmetries. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'18).* 1–7.
[19] D. Slepian. 1952. On the number of symmetry types of Boolean functions of n variables. *Can. J. Math.* (1952), 185–193.
[20] L. Benini and G. De Micheli. 1997. A survey of Boolean matching techniques for library binding. *ACM Trans. Des. Autom. Electron. Syst.* 2, 3 (1997), 193–226.
[21] U. Hinsberger and R. Kolla. 1998. Boolean matching for large libraries. In *Proceedings of the Design Automation Conference (DAC'98).* 206–211.
[22] D. Debnath and T. Sasao. 2004. Efficient computation of canonical form for Boolean matching in large libraries. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC'04).* 591–596.
[23] E. Mailhot and G. De Micheli. 1993. Algorithms for technology mapping based on binary decision diagrams and on Boolean operations. *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.* 12, 5 (1993), 599–620.
[24] E. M. Clarke, K. L. McMillan, X. Zhao, M. Fujita, and J. Yang. 1993. Spectral transforms for large Boolean functions with applications to technology mapping. In *Proceedings of the Design Automation Conference (DAC'93)*, 54–60.
[25] J. Rice. 2009. The autocorrelation transform and its application to the classification of Boolean functions. In *Proceedings of the IEEE Pacific Rim Conference on Communications Computers and Signal Processing (PacRim'09).* 94–99.
[26] K. Wang, C. Chan, and J. Liu. 2009. Simulation and SAT-based Boolean matching for large Boolean networks. In *Proceedings of the Design Automation Conference (DAC'09).* 396–401.
[27] C.-F. Lai, J.-H. R. Jiang, and K.-H. Wang. 2010. BooM: A decision procedure for Boolean matching with abstraction and dynamic learning. In *Proceedings of the Design Automation Conference (DAC'10).* 499–504.
[28] M. Soeken, A. Mishchenko, A. Petkovska, B. Sterin, P. Ienne, R. Brayton, and G. De Micheli. 2016. Heuristic NPN classification for large functions using AIGs and LEXSAT. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing (SAT'16).* 212–227.
[29] C. C. Tsai and M. Marek-Sadowaska. 1996. Generalized Reed-Muller forms as a tool to detect symmetries. *IEEE Trans. Comput.* 45, 1 (1996), 33–40.
[30] N. Kettle and A. King. 2006. An anytime symmetry detection algorithm for ROBDDs. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC'06)*, 24–27.
[31] V. N. Kravets and K. A. Sakallah. 2000. Generalized symmetries in Boolean functions. In *Proceedings of the International Conference on Computer Aided Design (ICCAD'00).* 526–532.
[32] J. Ciric and C. Sechen. 2003. Efficient canonical form for Boolean matching of complex functions in large libraries. *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.* 22, 5 (2003), 535–544.
[33] Q. Wu, C. Y. R. Chen, and J. M. Acken. 1994. Efficient Boolean matching algorithm for cell libraries. In *Proceedings of the International Conference on Computer Design.* 36–39.
[34] Selmer M. Johnson. 1963. Generation of permutations by adjacent transposition. *Math. Comp.* 17 (1963), 282–285.
[35] Berkeley Logic Synthesis and Verification Group. ABC: A System for Sequential Synthesis and Verification. Retrieved from http://www-cad.eecs. berkeley.edu/~alanmi/abc.
[36] R. Ashenhurst. 1957. The decomposition of switching functions. In *Proceedings of the International Symposium on the Theory of Switching.* Cambridge, Mass, 74–116.

Received December 2018; accepted February 2019