Generalized Boolean Symmetries Through Nested Partition Refinement

Hadi Katebi, Karem A. Sakallah, Igor L. Markov
School of Electrical Engineering and Computer Science
University of Michigan - Ann Arbor
Email: {hadik,karem,imarkov}@eecs.umich.edu

***Abstract*—Combinatorial objects in EDA applications exhibit a great amount of complexity and typically defy polynomial-time algorithms. To achieve acceptable performance, EDA tools seek to exploit various structures found in these objects in practice. In this work, we explore symmetries of Boolean functions and develop a new algorithm based on nested partition refinement, abstract group theory and Boolean satisfiability. We apply our algorithm to solve large-scale Boolean matching.**

I. Introduction

Finite combinatorial objects are viewed as subsets of power sets, Cartesian products, and derived sets. For example, a directed graph is defined by a set of vertices V and a set of edges $E \subseteq V \times V$, an undirected hypergraph is similarly defined by its hyperedges $E \subseteq 2^V$, and a Boolean function is defined by its input set X and minterms $M \subseteq 2^X$. In all such cases, there is a notion of underlying *variable set*, to which one can apply permutations or value substitutions.

A *symmetry* of a finite combinatorial object is a permutation of the object's variables that leaves the object unchanged. For example, a symmetry of Boolean function f is a permutation of f's inputs and outputs (with their possible negation) that preserves the value of f for all input combinations. A symmetry, under this *generalized* definition, is not limited to just a swap or simultaneous swaps of variables. Instead, it comprises a number of *rotations*, where a swap is a rotation with two variables. As an example, a 4-to-1 multiplexer exhibits 16 symmetries under the permutation and negation of its I/Os. These symmetries are listed in Fig. 1.[1]

MUX: $z = a_0 s_1' s_0' + a_1 s_1' s_0 + a_2 s_1 s_0' + a_3 s_1 s_0$
$\gamma_1 : \iota$
$\gamma_2 : (a_0, a_2)(a_0', a_2')(a_1, a_3)(a_1', a_3')(s_1, s_1')$
$\gamma_3 : (a_1, a_2)(a_1', a_2')(s_0, s_1)(s_0', s_1')$
$\gamma_4 : (a_0, a_1, a_3, a_2)(a_0', a_1', a_3', a_2')(s_0, s_1, s_0', s_1')$
$\gamma_5 : (a_0, a_1)(a_0', a_1')(a_2, a_3)(a_2', a_3')(s_0, s_0')$
$\gamma_6 : (a_0, a_3)(a_0', a_3')(a_1, a_2)(a_1', a_2')(s_0, s_0')(s_1, s_1')$
$\gamma_7 : (a_0, a_2, a_3, a_1)(a_0', a_2', a_3', a_1')(s_0, s_1', s_0', s_1)$
$\gamma_8 : (a_0, a_3)(a_0', a_3')(s_0, s_1')(s_0', s_1)$
$\gamma_9 : (a_0, a_0')(a_1, a_1')(a_2, a_2')(a_3, a_3')(z, z')$
$\gamma_{10} : (a_0, a_2')(a_0', a_2)(a_1, a_3')(a_1', a_3)(s_1, s_1')(z, z')$
$\gamma_{11} : (a_0, a_0')(a_1, a_2')(a_1', a_2)(a_3, a_3')(s_0, s_1)(s_0', s_1')(z, z')$
$\gamma_{12} : (a_0, a_1', a_3, a_2')(a_0', a_1, a_3', a_2)(s_0, s_1, s_0', s_1')(z, z')$
$\gamma_{13} : (a_0, a_1')(a_0', a_1)(a_2, a_3')(a_2', a_3)(s_0, s_0')(z, z')$
$\gamma_{14} : (a_0, a_3')(a_0', a_3)(a_1, a_2')(a_1', a_2)(s_0, s_0')(s_1, s_1')(z, z')$
$\gamma_{15} : (a_0, a_2', a_3, a_1')(a_0', a_2, a_3', a_1)(s_0, s_1', s_0', s_1)(z, z')$
$\gamma_{16} : (a_0, a_3')(a_0', a_3)(a_1, a_1')(a_2, a_2')(s_0, s_1')(s_0', s_1)(z, z')$

Fig. 1. Symmetries of a 4-to-1 MUX under permutation/negation of I/Os.

Generating Set: $g = \{\gamma_3, \gamma_5, \gamma_9\}$

$\gamma_1 = \gamma_3 \cdot \gamma_3$	$\gamma_2 = \gamma_3 \cdot (\gamma_5 \cdot \gamma_3)$	$\gamma_4 = \gamma_5 \cdot \gamma_3$
$\gamma_6 = \gamma_5 \cdot (\gamma_3 \cdot (\gamma_5 \cdot \gamma_3))$	$\gamma_7 = \gamma_3 \cdot \gamma_5$	$\gamma_8 = \gamma_5 \cdot (\gamma_3 \cdot \gamma_5)$
$\gamma_{10} = (\gamma_3 \cdot (\gamma_5 \cdot \gamma_3)) \cdot \gamma_9$	$\gamma_{11} = \gamma_3 \cdot \gamma_9$	$\gamma_{12} = (\gamma_5 \cdot \gamma_3) \cdot \gamma_9$
$\gamma_{13} = \gamma_5 \cdot \gamma_9$	$\gamma_{14} = (\gamma_5 \cdot (\gamma_3 \cdot (\gamma_5 \cdot \gamma_3))) \cdot \gamma_9$	
$\gamma_{15} = (\gamma_3 \cdot \gamma_5) \cdot \gamma_9$	$\gamma_{16} = (\gamma_5 \cdot (\gamma_3 \cdot \gamma_5)) \cdot \gamma_9$	

Fig. 2. A generating set for the symmetry group of the MUX in Fig. 1.

The set of all symmetries of an object forms a *group*[2] under functional composition. This group is referred to as the *symmetry group* of the object. In general, the size of the symmetry group of an object is *exponential* in the number of its variables. Nevertheless, all symmetries of an object can be generated from just a subset of its symmetries. This is accomplished by repeatedly composing the elements of that subset under functional composition. Such a subset is called a *symmetry group generating set*. Fig. 2 shows a 3-element symmetry group generating set for the 4-to-1 multiplexer of Fig. 1, and shows how the remaining 13 symmetries of the multiplexer can be generated from that generating set.

In this paper, we study the symmetries of Boolean functions, which find numerous applications in logic synthesis and verification. One common application is in *Boolean matching*, where functional equivalence of Boolean functions under permutation (and negation) of inputs and outputs is investigated [1], [2]. Other applications include BDD minimization [3] and circuit power optimization [4].

Most existing symmetry-detection algorithms for Boolean functions only look for *classical symmetries*, i.e., symmetries that include just *a single swap* of variables [5], [3]. As the number of such symmetries is at most quadratic in the number of a function's inputs, they can be evaluated one by one and explicitly enumerated. The caveat of these algorithms, however, is that they overlook symmetries that involve more than two variables. For instance, none of the 16 symmetries of the multiplexer in Fig. 1 would be found this way.

Higher order symmetries, formed by *simultaneous swaps* of variables, have also been addressed in the literature. The algorithm in [6] captures higher order symmetries (under permutation and negation of inputs) by performing *hierarchical partitioning* on the set of variables of a netlist. This algorithm, although capable of reporting symmetries beyond classical, does not always find all symmetries of a Boolean function.

Furthermore, most symmetry-detection algorithms only allow permutation of inputs, but not permutation of outputs [5], [6]. Such algorithms report symmetries of a multi-output function by isolating

[1] A rotational symmetry of the form $(a_1, a_2, a_3, ..., a_n)$ maps a_1 to a_2, a_2 to a_3, ..., and a_n to a_1. Also, ι denotes the identity.

[2] Group theory is a branch of abstract algebra that studies the algebraic structures known as *groups*. A *group* comprises a non-empty set of elements with a binary operation that is *associative*, admits an *identity* element, and is *invertible*. For example, the set of integers with addition forms a group.

each output one at a time. Nevertheless, symmetries that are formed by simultaneous permutations of inputs and outputs are beneficial in EDA. For instance, [4] uses such symmetries to enhance post-placement algorithms. It, however, performs an exhaustive search for symmetries, and hence, can only handle small (sub)circuits.

In this paper, we propose a new algorithm for detecting symmetries of Boolean functions under permutation but not negation of I/Os (we plan to modify the proposed algorithm to consider negation of I/Os in our future work). Our algorithm takes a function in the form of an *And-Inverter graph* (*AIG*), constructs a complete permutation tree, and systematically prunes it by integrating group-theoretic (and other) techniques. To accomplish this, it builds several graphs based on functional dependency and random simulation, and uses them to refine the search space. It also takes advantage of satisfiability to test functional equivalence under candidate permutations, and learns from satisfiability counterexamples to avoid recurring conflicts.

We integrate our algorithm within the ABC package [7] — an established system for synthesis and verification of logic circuits. We test the performance of our algorithm on a collection of available combinational circuits. As part of our study, we also encode Boolean matching as a symmetry-detection problem, and report the results of applying our algorithm to several Boolean matching instances.

Key contributions of our work include:

1) Proposing a novel symmetry-detection algorithm for Boolean functions based on group-theoretic concepts.
2) Allowing permutations of both inputs and outputs.
3) Learning from satisfiability counterexamples.
4) Formulating Boolean matching as symmetry detection.

The remainder is organized as follows. Section II provides definitions and notation for Boolean functions and discusses relevant work. Section III describes our symmetry-discovery algorithm for Boolean functions. Section IV formulates Boolean matching as a symmetry-detection problem. Section V presents experimental results. Section VI provides conclusions and discusses future work.

II. Definitions and Background

This section provides definitions and notation for Boolean functions and their symmetries. It also discusses relevant background.

A. Boolean Functions and Related Definitions

An n-input m-output *Boolean function* f is a function on B^n into B^m, where $B = \{0, 1\}$. We denote the set of all inputs of f by $X = \{x_1, ..., x_n\}$, and the set of all outputs of f by $Z = \{z_1, ..., z_m\}$.

An input vector $P = \langle p_1, ..., p_n \rangle$ of f assigns value $p_i \in \{0, 1\}$ to input $x_i \in X$. The output vector $R = \langle r_1, \cdots, r_m \rangle$ that corresponds to input vector P is the result of simulating P with f, where $r_i \in \{0, 1\}$ holds the simulation result for output $z_i \in Z$. Input $x_i \in X$ is *observable* to output $z_j \in Z$ (or output z_j is *controllable* by input x_i) with regard to input vector $P = \langle p_1, ..., p_n \rangle$ and its corresponding output vector $R = \langle r_1, \cdots, r_m \rangle$, if flipping $p_i \in P$ flips $r_j \in R$.

The *positive (negative) cofactor* of f with regard to input $x \in X$, denoted by f_x ($f_{x'}$), is the function that fixes the value of x to one (zero). The *support* of output $z \in Z$, denoted by $supp(z)$, is the set of all inputs $x \in X$ that functionally affect z, i.e, $supp(z) = \{x \in X \mid f_x \neq f_{x'} \text{ for } z\}$.

A *partition* $\pi = [W_1|W_2|\cdots|W_t]$ of a set is a list of non-empty pair-wise disjoint subsets of the set whose union is the set. An *ordered partition* is a partition whose subsets are ordered. The subsets W_i are called the *cells* of the partition. The size of cell W_i is denoted by $|W_i|$.

An *ordered partition pair* (*OPP*) Π of a set is specified as

$$\Pi = \begin{bmatrix} \pi_T \\ \pi_B \end{bmatrix} = \begin{bmatrix} T_1 \,|T_2\,|\cdots|T_s \\ B_1\,|B_2\,|\cdots|B_t \end{bmatrix}$$

with π_T and π_B referred to, respectively, as the top and bottom ordered partitions of Π. OPP Π is *isomorphic* if $s = t$ and $|T_i| = |B_i|$ for $i = 1, \cdots, t$; otherwise *non-isomorphic*. An isomorphic OPP is *discrete* if $|T_i| = |B_i| = 1$ for $i = 1, \cdots, t$, and *unit* if $s = t = 1$.

Input vector $P = \langle p_1, ..., p_n \rangle$ is said to be *proper* with regard to partition $\pi = [W_1|\cdots|W_t]$ of input set X, if it assigns the same value to all inputs in the same cell of π, i.e., for all i and j, $p_i = p_j$ if $x_i, x_j \in W_l$, for some l. Two input vectors $P = \langle p_1, ..., p_n \rangle$ and $Q = \langle q_1, ..., q_n \rangle$ are said to be *consistent* with regard to isomorphic OPP of input set X

$$\Pi = \begin{bmatrix} \pi_T \\ \pi_B \end{bmatrix} = \left[\begin{array}{c|c|c|c} T_1 & T_2 & \cdots & T_s \\ B_1 & B_2 & \cdots & B_s \end{array}\right] \quad (1)$$

if P is proper with regard to π_T, Q is proper with regard to π_B, and P and Q assign the same value to all inputs in the same-index cells of π_T and π_B, i.e., for all i and j, $p_i = q_j$ if $x_i \in T_l$ and $x_j \in B_l$, for some l.

B. Symmetry in Boolean Functions

We assume familiarity with basic notions from group theory, including such concepts as *groups*, *subgroups*, *group generators*, *cosets*, *orbit partition*, etc. More information on these concepts is available in abstract algebra texts such as [8].

Given Boolean function f with input set X and output set Z, a *permutation* of f is defined as a bijection from X to X and Z to Z. Permutation γ, when applied to f, permutes f's inputs and outputs, and produces function f^γ.

A *symmetry* (*automorphism*) of f is a permutation γ of f that preserves f's functionality, i.e., $f^\gamma = f$, where "=" denotes functional equivalence. Every function has a trivial symmetry, called the *identity* (denoted by ι), that maps each I/O to itself. Two functions f_1 and f_2 are *isomorphic* if and only if there exists a permutation γ such that $f_1^\gamma = f_2$.

The set of all symmetries of f forms a *group* under functional composition. This group is called the *symmetry group* of f, and is denoted by $\mathcal{G}$. A *generating set* for $\mathcal{G}$ is a subset of the symmetries in $\mathcal{G}$ whose combinations under functional composition generate $\mathcal{G}$. Each element of a generating set is called a (*group*) *generator*.

A *subgroup* of $\mathcal{G}$ is a subset of $\mathcal{G}$ that forms a group under functional composition. The *stabilizer subgroup* of I/O $i \in X \cup Z$, denoted by $\mathcal{G}_i$, is a subgroup of $\mathcal{G}$ that fixes i, i.e., $\mathcal{G}_i = \{\gamma \in \mathcal{G} \mid \gamma i = i\}$.

The (*right*) *coset* of $\mathcal{G}_i$ in $\mathcal{G}$ containing $\sigma \in \mathcal{G}$ is the set $\{\gamma\sigma \mid \gamma \in \mathcal{G}_i\}$. The set of all cosets of $\mathcal{G}_i$ in $\mathcal{G}$ partitions $\mathcal{G}$ into equally-sized subsets. Choosing one element from each coset yields a set of *coset representatives*. Each coset representative composed with $\mathcal{G}$ generates the entire coset.

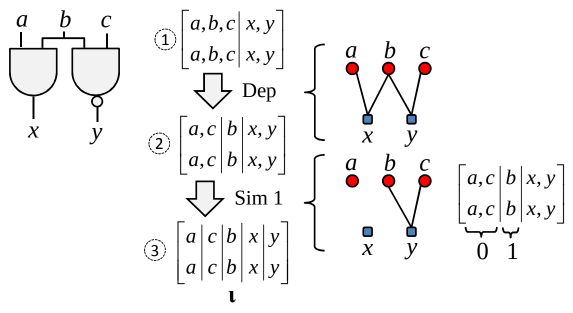


Fig. 3. An example function with 1 symmetry.

Given $\mathcal{G}$, $i \sim j$ for inputs $i, j \in X$ or outputs $i, j \in Z$ (read i and j share the same *orbit*), if and only if there exists symmetry $\gamma \in \mathcal{G}$ that maps i to j, i.e., $\gamma i = j$. "$\sim$" defines an *equivalence* relation on set $X \cup Z$, which partitions $X \cup Z$ into a so-called *orbit partition* Θ. Each cell of the orbit partition is called an *orbit*. The orbit that contains i is written as Θ_i.

C. And-Inverter Graphs

An *And-Inverter graph* (*AIG*) is a directed acyclic graph that represents the functionality of a Boolean function. The nodes of an AIG are two-input "And" gates, and its edges are optionally marked to indicate "Not" gates. Modern logic synthesis tools, such as ABC, use AIGs as alternatives to *Binary Decision Diagrams* (*BDDs*), since AIGs are more memory efficient, and are faster in performing logic simulation. Unlike BDDs, AIGs are not canonical, but are *structurally hashed* to be partially canonical [9]. Without limiting our work, we assume that circuits are represented by structurally hashed AIGs, .

D. Boolean Satisfiability

Boolean satisfiability (*SAT*) seeks a variable assignment to a Boolean function that evaluates the function to true. If such an assignment exists, the function is *satisfiable*, and *unsatisfiable* otherwise. One application of SAT in EDA is to check the *functional equivalence* of two combinational circuits [10]. This is accomplished by building the *miter* of the two circuits and passing it to a SAT solver. The miter of two circuits is constructed by combining inputs with the same name, feeding outputs with the same name to two-input XOR gates, and connecting the outputs of the XOR gates to one multi-fanin OR gate. If the miter is unsatisfiable, the circuits are equivalent. If not, the circuits are not equivalent, and the satisfiable assignment serves as a counterexample. In our work, we use SAT to check the equivalence of a Boolean function under permutations of its I/Os.

III. Proposed Symmetry-Detection Algorithm for Boolean Functions

In this section, we describe our algorithm for detecting symmetries of Boolean functions under permutation of I/Os. We periodically refer to the three examples depicted in Fig. 3, Fig. 4, and Fig. 5 to illustrate our algorithm.

A. Implicit Representation of Permutation Sets

OPPs play a central role in our symmetry-detection algorithm, since they provide a compact implicit representation of sets of permutations. Specifically, a discrete OPP represents a single permutation, whereas a unit OPP represents all $k!$ permutations of a

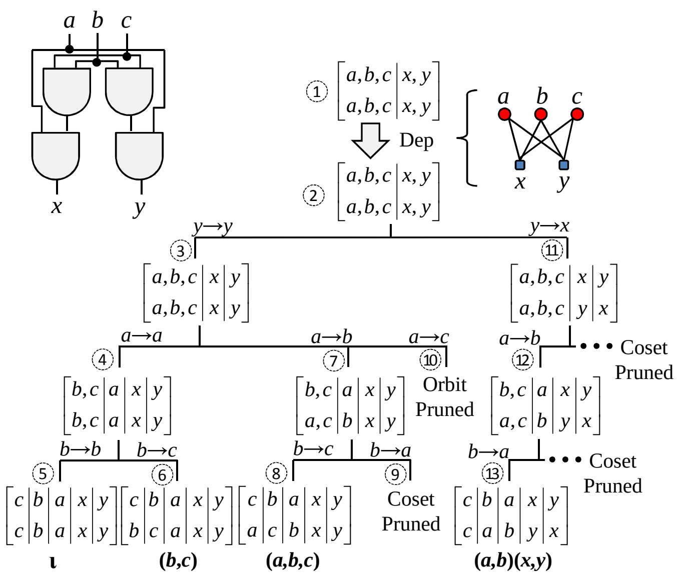


Fig. 4. An example function with 12 symmetries.

k-variable object. In general, the isomorphic OPP in (1) represents $\prod_{1\le i\le s} |T_i|!$ permutations. On the other hand, note that it is not possible to obtain well-defined mappings between the top and bottom partitions of a non-isomorphic OPP. Thus, non-isomorphic OPPs serve as empty sets of permutations. Below are several example OPPs and the permutation sets they encode:

$$-\text{Discrete OPP: } \left[\begin{array}{c|c|c} x_1 & x_2 & x_3 \\ x_2 & x_3 & x_1 \end{array}\right] = \{(x_1, x_2, x_3)\}$$

$$-\text{ Unit OPP: } \left[\begin{array}{c} x_1, x_2, x_3 \\ x_1, x_2, x_3 \end{array}\right] = \{\iota, (x_1, x_2), (x_1, x_3), (x_2, x_3), (x_1, x_2, x_3), (x_1, x_3, x_2)\}$$

$$-\text{ Isomorphic OPP: } \left[\begin{array}{c|c} x_3 & x_1, x_2 \\ x_2 & x_3, x_1 \end{array}\right] = \{(x_2, x_3), (x_1, x_3, x_2)\}$$

$$-\text{ Non-isomorphic OPPs: } \left[\begin{array}{c} x_1, x_3 | \, x_2 \\ x_2 | \, x_3, x_1 \end{array}\right] = \emptyset$$

B. Basic Enumeration of the Permutation Search Space

To search for the symmetries of a Boolean function, our algorithm builds a permutation tree whose nodes are OPPs. Each OPP represents a set of I/O permutations. The OPP at the root, for example, is an isomorphic OPP that gathers all inputs (resp. outputs) in the same-index cells of the top and bottom partitions. This OPP encodes all $n!m!$ I/O permutations.

The basic skeleton of our permutation enumeration algorithm is formed by extending isomorphic OPPs using the following procedure:

- choosing a non-singleton cell (the *target* cell) from the top partition,
- choosing an I/O from the target cell (the *target* I/O),
- mapping the target I/O to an I/O (the *candidate* I/O) from the corresponding cell of the bottom partition.

The mapping step splits the target cell so that the target I/O has its own cell. It then similarly splits the candidate I/O in the bottom partition by putting it in its cell under the target I/O. For example, node (11) of the tree in Fig. 4 maps y to x, and hence, the OPP at node (11) separates y from x on the top and x from y on the bottom.

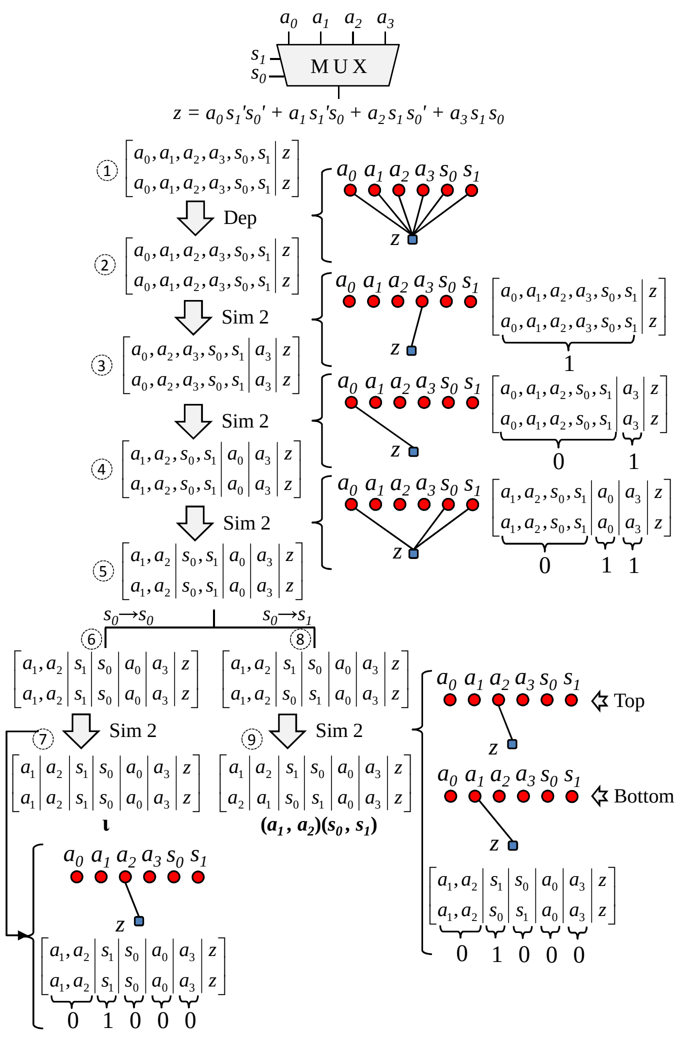


Fig. 5. A 4-to-1 multiplexer which has 2 symmetries.

The permutation search tree can be pruned significantly by performing *partition refinement* before branching on a target I/O. This is accomplished by modeling *partial* functionality of the Boolean function by several *abstraction graphs*, and using them to preclude permutations that do not yield any symmetry. In the trees of Fig. 3, Fig. 4, and Fig. 5, down arrows refer to refinement steps. The abstraction graph used by each refinement is shown next to it. We discuss partition refinement in detail in Sections III-C and III-D. Furthermore, our tree exploits two *group-theoretic* pruning techniques, namely, *coset* and *orbit* pruning, to avoid exploring an exponential space of symmetries. These two techniques are explained in Section III-E.

After partition refinement, an OPP is either isomorphic or non-isomorphic. A non-isomorphic OPP reflects a *conflict*. A discrete OPP, however, is a candidate for symmetry which needs to be verified by SAT (see Section III-F). If SAT disqualifies a symmetry, it returns a counterexample which is used to further prune the search (see Section III-G). Mapping I/Os continues until a conflict or a candidate symmetry is detected. In either case, our algorithm backtracks and maps the target I/O at the backtrack level to the remaining candidate I/Os. The search ends when all possible mappings are exhausted.

It should be mentioned that our enumeration algorithm first maps outputs and then inputs. Our experiments show that once outputs are distinguished, inputs can be distinguished quickly.

C. Abstraction Graphs

Abstraction graphs are two-colored bipartite graphs constructed to partially capture the functionality of a Boolean function. These graphs are used by partition refinement to prune the permutation space. Here, we introduce three types of abstraction graphs; one based on functional dependency, and two based on random simulation.

1) Dependency Graph: The *dependency graph* of a Boolean function encodes the supports of the function as a graph using the following procedure. A red vertex is added for each input, and a blue vertex for each output. An edge is added between input x and output z if and only if $x \in supp(z)$. For the functions of Fig. 3, Fig. 4, and Fig. 5, the dependency graphs are depicted where refinement is labeled with “Dep”.

We build dependency graphs to distinguish outputs (resp. inputs) that have different functional dependency (resp. influence). For example, at node (2) of the tree in Fig. 3, input b is separated from inputs a and c, since the degree of b in the dependency graph is different from that of a and c.

2) Simulation Graphs: We construct two types of simulation graphs based on proper random input vectors. Intuitively, a proper random input vector assigns the same value to all inputs that are not yet distinguished. For the functions of Fig. 3, Fig. 4, and Fig. 5, type-1 and type-2 simulation graphs are depicted where refinement is labeled with “Sim 1” and “Sim 2”, respectively. These graphs are built based on proper input vectors that are shown next to them.

Given a proper input vector $P = \langle p_1, ..., p_n \rangle$ (with regard to partition π of input set X) and its corresponding output vector $R = \langle r_1, ..., r_m \rangle$, we build two types of simulation graphs as follows:

– *Simulation Graph Type 1*: We add a red vertex for each input, and a blue vertex for each output. We add an edge between $z_i \in Z$ and all inputs $x \in supp(z_i)$, if and only if $r_i = 1$. For example, the simulation graph at node (2) of the tree in Fig. 3 encodes the fact that if $a = c = 0$ and $b = 1$, then $x = 0$ and $y = 1$.

– *Simulation Graph Type 2*: We add a red vertex for each input, and a blue vertex for each output. We then flip each $p_i \in P$, one at a time, and save the resulting n input vectors in $P_1, ..., P_n$. We simulate $P_1, ..., P_n$ and record the resulting n output vectors in $R_1 = \langle r_1^1, ..., r_m^1 \rangle, ..., R_n = \langle r_1^n, ..., r_m^n \rangle$. We add an edge between $z_j \in Z$ and $x_i \in X$, if and only if $r_j^i \neq r_j$. For example, the simulation graph at node (7) of the tree in Fig. 5 encodes the fact that if $a_1 = a_2 = s_0 = a_0 = a_3 = 0$ and $s_1 = 1$, flipping a_2 flips z.

We build type-1 simulation graphs primarily to distinguish outputs. Once outputs are distinguished, inputs might be distinguished as well. For example, at node (3) of the tree in Fig. 3, output y is separated from output x, and subsequently, input c is separated from input a. Furthermore, we build type-2 simulation graphs to distinguish inputs (outputs) that have different observability (controllability). For example, at node (7) of the tree in Fig. 5, input a_2 is separated from input a_1, since the observability of a_2 is different from that of a_1 (with regard to the given random input vector).

D. Refinement by Abstraction Graphs

Partition refinement based on an abstraction graph propagates the constraints (i.e., vertex colors, vertex degrees, and edge relation) of the graph until the partition becomes *equitable*. Partition

```
Inputs: $\Pi$, $G_D$
Outputs: $L_1$, $L_2$
1) Set $\pi_T$ to the top partition of OPP $\Pi$.
2) Refine partition $\pi_T$ by dependency graph $G_D$.
3) Build ordered partition $\pi = [W_1|\cdots|W_t] \subset \pi_T$ by removing
   the cells of $\pi_T$ that contain the outputs.
4) Set counter $i = 0$.
5) Generate random bit vector $B = \langle b_1, ..., b_t\rangle$, where $b_i \in \{0, 1\}$.
6) Generate proper random input vector $P = \langle p_1, ..., p_n\rangle$ with
   regard to $\pi$, where $p_i = b_j$ if input $x_i \in W_j$.
7) Generate type-1 simulation graph $G$ that corresponds to input
   vector $P$.
8) Refine $\pi_T$ by simulation graph $G$.
9) If new cells are induced at line 8, refine $\pi_T$ by dependency graph
   $G_D$, save bit vector $B$ in list $L_1$, and set counter $i = 0$.
10) Increment $i$. Go to line 5 if $i < 200$.
11) Repeat lines 4-10, but this time generate type-2 simulation graph
    $G$ at line 7, and save bit vector $B$ in list $L_2$ at line 9.
```

Fig. 6. Pseudocode for refining the top partition of OPP Π.

```
Inputs: $\Pi$, $G_D$, $L_1$, $L_2$
Outputs: 0 or 1
1) Set $\pi_B$ to the bottom partition of OPP $\Pi$.
2) Refine partition $\pi_B$ by dependency graph $G_D$.
3) Build ordered partition $\pi = [W_1|\cdots|W_t] \subset \pi_B$ by removing
   the cells of $\pi_B$ that contain the outputs.
4) Set counter $i = 0$.
5) Set bit vector $B = \langle b_1, ..., b_t\rangle$ to the $i$-th element of list $L_1$.
6) Generate proper random input vector $P = \langle p_1, ..., p_n\rangle$ with
   regard to $\pi$, where $p_i = b_j$ if input $x_i \in W_j$.
7) Generate type-1 simulation graph $G$ that corresponds to input
   vector $P$.
8) Refine $\pi_B$ by simulation graph $G$. Return 0 if a conflict is
   detected.
9) Refine $\pi_B$ by dependency graph $G_D$. Return 0 if a conflict is
   detected.
10) Increment $i$. Go to line 5 if $i < size(L_1)$.
11) Repeat lines 4-10, but this time set bit vector $B$ to the $i$-th
    element of list $L_2$ at line 5, generate type-2 simulation graph
    $G$ at line 7, and check $i < size(L_2)$ at line 10.
12) Return 1.
```

Fig. 7. Pseudocode for refining the bottom partition of OPP Π.

$\pi = [W_1|W_2|\cdots|W_t]$ is said to be equitable (with respect to a given graph) if, for all vertices $v_1, v_2 \in W_i$ $(1 \leq i \leq t)$, the number of neighbors of v_1 in W_j $(1 \leq j \leq t)$ is equal to the number of neighbors of v_2 in W_j.

In our context, refinement is applied *simultaneously* [11] to the top and bottom partition of an OPP, until 1) both partitions become equitable and the resulting OPP is isomorphic, or 2) the resulting OPP is non-isomorphic indicating an empty set of permutations (i.e., a conflict). In implementation, our algorithm first refines the top partition until it becomes equitable, and records where the cell splits occur. Then, it starts refining the bottom partition, and compares the splitting locations of the bottom to the top (i.e., checks the isomorphism of the two partitions) after each refinement step. It also ensures that the connections of each newly created cell on the bottom match the connections of its corresponding cell on the top.

Fig. 6 and Fig. 7 demonstrate the refinement routines for the top and bottom partitions, respectively. In these figures, "Refine" refers to the simultaneous partition refinement explained above.

The routine of Fig. 6 primarily refines the top partition by the dependency graph (lines 1-2). It then generates a number of proper input vectors with regard to the subset of the top partition that includes just the inputs of the function (lines 3-6 and 10). Next, it builds type-1 simulation graphs based on the generated input vectors (line 7), and uses them to refine the top partition (line 8). It refines once more by dependency graph if new cells were induced at the previous step (line 9). It also saves the bit vectors whose corresponding simulation graphs caused further refinement (line 9). These vectors will later be used by the refinement of the bottom partition to generate consistent input vectors. This routine ends by following similar refinement steps for type-2 simulation graphs (line 11).

The routine of Fig. 7 resembles that of Fig. 6, but with two main differences. First, it does not generate new random bit vectors. Instead, it uses the ones generated by Fig. 6 to make pairs of consistent input vectors (lines 5-6). In other words, it assigns the same Boolean value to all potentially mappable inputs of the top and bottom partitions. Second, it checks OPP isomorphism during refinement, and returns 0 if a conflict is detected (lines 8-9).

In the tree of Fig. 5, refinement at node (8) assigns the same Boolean value to all inputs in the same-index cells of the top and bottom partitions. It then refines the OPP using type-2 simulation graph. The result of refinement is the isomorphic OPP at node (9).

E. Group-Theoretic Pruning

Our symmetry-detection algorithm exploits two group-theoretic pruning techniques: *coset pruning* and *orbit pruning*. These techniques are standard in high-performance symmetry-detection packages to avoid enumeration of an exponential number of symmetries.

Coset pruning affirms that one generator per coset (i.e., the coset representative) can generate all symmetries in the coset. In fact, it allows our algorithm to look for *one* generator per coset. For example, node (9) of the tree in Fig. 4 is coset pruned, since the symmetry at node (8) is a coset representative for the coset at node (7).

Orbit pruning uses the orbit partition to eliminate redundant generators. It exempts our algorithm from seeking symmetries that can be generated from already found symmetries. For example, node (10) of the tree in Fig. 4 is orbit pruned, since composing the symmetries at nodes (8) and (6) yields a new symmetry that maps input a to input c.

Coset and orbit pruning techniques are enabled in our algorithm because the left-most path of our search tree corresponds to a sequence of *subgroup stabilizers*, ending in the identity. In other words, "decisions" along that path map each I/O to itself. This requirement does not apply to decisions in other parts of the tree.

It should be noted that our algorithm finds at most $n + m - 2$ symmetry group generators for an n-input m-output Boolean function. It also calculates the size of the symmetry group using the orbit-stabilizer and Lagrange theorems [8]: $|\mathcal{G}| = |\mathcal{G}_i| \cdot |\Theta_i|$.

F. Checking Functional Equivalence by SAT

A candidate symmetry (returned by refinement) needs to be verified by SAT, since refinement by abstraction graphs per se does not prove functional equivalence. To perform this verification, our algorithm permutes the I/Os of the function according to the candidate symmetry, builds the miter of the original and permuted functions,

```
Inputs: $\gamma$, $f$, $C$, $DB$
Outputs: None
1) Set input vectors $P$ to SAT counterexample $C$.
2) Set input vectors $Q$ to SAT counterexample $C$.
3) Permute input vector $Q$ based on permutation $\gamma$.
4) Simulate function $f$ with input vector $P$ and save the simulation
   result in output vector $R$.
5) Simulate function $f$ with input vector $Q$ and save the simulation
   result in output vector $U$.
6) Build simulation pairs $\langle P, R\rangle$ and $\langle Q, U\rangle$.
7) Check if simulation pair $\langle P, R\rangle$ already exists in database $DB$.
   If no, add $\langle P, R\rangle$ to $DB$, and set the activity of $\langle P, R\rangle$ to 0.
8) Repeat line 7, but this time for simulation pair $\langle Q, U\rangle$.
9) If $size(DB) > 50$, reduce $DB$.
```

Fig. 8. Pseudocode for learning from a SAT counterexample.

and hands off the miter to SAT. If SAT finds the miter unsatisfiable, a symmetry is found; otherwise, a *functional conflict* is detected, and a counterexample is returned. In Fig. 3, Fig. 4, and Fig. 5, all discrete OPPs form symmetries of the corresponding functions.

G. Learning From SAT Counterexamples

Partition refinement typically reduces the number of possible matches from $n!m!$ to hundreds or less, often making exhaustive search (with SAT-based equivalence checking) practical. However, this phase of search can be significantly improved by learning from SAT counterexamples.

A SAT counterexample is in the form of an input vector that forces the miter of the original function and the permuted function to be satisfiable. Our algorithm learns a collection of such input vectors, along with their corresponding output vectors, and uses them to backjump to the tree level where functional conflicts are resolved.

Fig. 8 shows the routine that learns from a SAT counterexample. This routine makes two copies of the counterexample (lines 1-2), and permutes one copy based on the candidate symmetry (line 3). It then simulates the function with the two copies, and saves the results in simulation pairs of the form $\langle$input vector, output vector$\rangle$ (lines 4-6). It attaches the two simulation pairs to the *database of simulation pairs*, and set their *activities* to zero (lines 7-8). The activity of a simulation pair quantifies its participation in conflict detection. This routine ends by potentially *reducing* the database (line 9) by finding the median of the activities of all simulation pairs and deleting pairs whose activities fall below the median.

Once a functional conflict is detected, our backjumping routine backtracks one level up at a time, checks the current OPP for functional conflicts, and stops backtracking once the OPP is found free of conflicts.

Fig. 9 shows the routine that checks for a functional conflict in an OPP. This routine searches the database of simulation pairs for two consistent input vectors (lines 1-13). Suppose that it finds input vectors P and Q consistent with regard to the OPP, and suppose that $\langle P, R\rangle$ and $\langle Q, U\rangle$ are the simulation pairs that correspond to P and Q. This routine counts the number of outputs in each cell of the top (resp. bottom) partition whose values in R (resp. U) is one (lines 14-15). It declares a conflict, if two same-index cells of the top and bottom partitions have different counters (line 16). In fact, it anticipates that such a pair of cells will eventually map two outputs whose simulation values, and hence, functional behaviors, are

```
Inputs: $\Pi$, $DB$
Outputs: 0 or 1
1) Set ordered partition $\pi_T$ and ordered partition $\pi_B$ to the top and
   bottom partitions of OPP $\Pi$, respectively.
2) Build two ordered partitions $\pi_T^i$ and $\pi_T^o$ from partition $\pi_T$, where
   $\pi_T^i$ contains cells of $\pi_T$ that have inputs, and $\pi_T^o$ contains cells
   of $\pi_T$ that have outputs.
3) Repeat line 2, but this time for partition $\pi_B$, and save the
   resulting sub-partitions in $\pi_B^i$ and $\pi_B^o$.
4) Set counter $i = 0$.
5) Return 1 if $i \geq size(DB)$.
6) Set simulation pair $\langle P, R\rangle$ to the $i$-the element of database $DB$.
7) Check if input vector $P$ is proper with regard to partition $\pi_T^i$. If
   no, increment $i$, and go to line 5.
8) Set counter $j = 0$.
9) Check $j \geq size(DB)$. If yes, increment $i$ and go to line 5.
10) Set simulation pair $\langle Q, U\rangle$ to the $j$-the element of database $DB$.
11) Check if input vector $Q$ is proper with regard to partition $\pi_B^i$.
    If no, increment $j$, and go to line 9.
12) Build OPP $\Pi^i$ by putting partitions $\pi_T^i$ and $\pi_B^i$ as the top and
    bottom partitions of $\Pi^i$, respectively.
13) Check if input vectors $P$ and $Q$ are consistent with regard to
    $\Pi^i$. If no, increment $j$, and go to line 9.
14) Set cell $C_k$ to the $k$-th cell of $\pi_T^o$. Set $N_k$ to the number of
    outputs in $C_k$ whose value in $R$ is 1. Do this for all $1 \leq k \leq$
    $size(\pi_T^o)$.
15) Set cell $C_k$ to the $k$-th cell of $\pi_B^o$. Set $M_k$ to the number of
    outputs in $C_k$ whose value in $U$ is 1. Do this for all $1 \leq k \leq$
    $size(\pi_B^o)$.
16) Check if $N_k \neq M_k$ for some $k$. If yes, increase activity of pairs
    $\langle P, R\rangle$ and $\langle Q, U\rangle$, and return 0.
17) Return 1.
```

Fig. 9. Pseudocode for checking functional conflicts in isomorphic OPP Π.

different under P and Q. At the end, it increases the activity of $\langle P, R\rangle$ and $\langle Q, U\rangle$ to credit their participation in conflict detection (line 16).

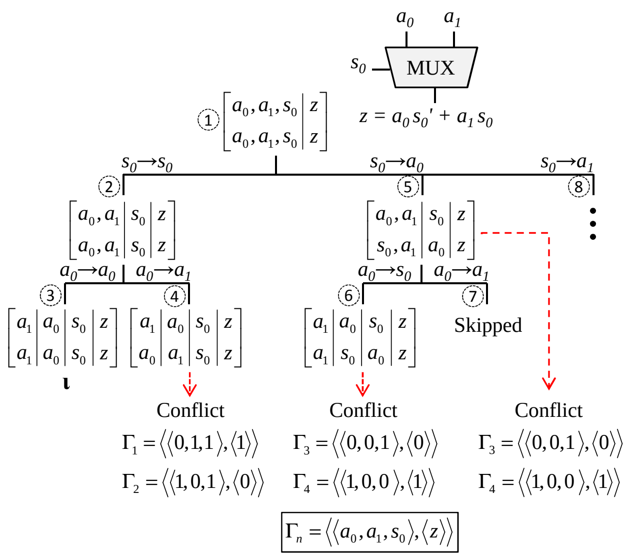
Fig. 10 shows an example of learning when refinement is disabled.[3] For this example, our algorithm encounters functional conflicts at nodes (4) and (6). While backtracking from node (6), it finds that node (5) has a functional conflict under simulation pairs Γ_3 and Γ_4. Hence, it backtracks from node (6) to node (8), and skips node (7).

IV. Case Study: Boolean Matching

PP-equivalence checking is a Boolean matching problem which seeks functional equivalence of two functions under permutation of I/Os. In other words, it checks the isomorphism of the two functions. Here, we explain how our symmetry-detection algorithm can be modified to solve the PP-equivalence checking problem.

An automorphism of a function is an isomorphism with itself. Hence, one can check isomorphism of two functions by putting them side by side, and passing them to an automorphism-detection algorithm that incorporates the two following modifications. First, the modified algorithm only needs to look for permutations that map I/Os of one function to another. In other words, it immediately prunes subtrees that (partially) map one function to itself. Second, it only needs to find one symmetry to confirm isomorphism. It should be mentioned that this algorithm does not use coset or orbit pruning, since it terminates the search as soon as it finds one symmetry.

[3]A search tree that could illustrate learning and perform partition refinement was too large to fit in the paper.



a_0 a_1
s_0
MUX
$z = a_0 s_0' + a_1 s_0$
$s_0 \to s_0$
$s_0 \to a_0$
$s_0 \to a_1$
$a_0 \to a_0$
$a_0 \to a_1$
$a_0 \to s_0$
$a_0 \to a_1$
Skipped
Conflict
Conflict
Conflict
$\Gamma_1 = \langle\langle 0,1,1\rangle,\langle 1\rangle\rangle$
$\Gamma_2 = \langle\langle 1,0,1\rangle,\langle 0\rangle\rangle$
$\Gamma_3 = \langle\langle 0,0,1\rangle,\langle 0\rangle\rangle$
$\Gamma_4 = \langle\langle 1,0,0\rangle,\langle 1\rangle\rangle$
$\Gamma_3 = \langle\langle 0,0,1\rangle,\langle 0\rangle\rangle$
$\Gamma_4 = \langle\langle 1,0,0\rangle,\langle 1\rangle\rangle$
$\Gamma_n = \langle\langle a_0,a_1,s_0\rangle,\langle z\rangle\rangle$

Fig. 10. An example function with its partial search tree built by our symmetry-detection algorithm when refinement is disabled.

V. Empirical Validation

We integrated our proposed symmetry-detection algorithm for Boolean functions in the ABC package (command `saucy3`) [7]. Our experiments were conducted on an HP workstation equipped with a 3.2GHz Intel Quad-Core CPU, an 8MB cache and an 8GB RAM, running 64-bit Windows 7. A time-out of 2000 seconds was applied.

Table I demonstrates the results of applying our symmetry-detection algorithm on a collection of benchmarks from ISCAS'85 [12], ISCAS'89 [13], MCNC [14], and ITC'99 [15]. In this table, the first column lists the name of the benchmarks. The next three columns list the number of inputs, number of outputs, and the size of AIG for each benchmark, respectively. Column #Symms shows the order of symmetry group, and Column #Gen shows the number of generators. Information on constructed search trees, such as the number of nodes, number of levels, and number of conflicts, are drawn in Columns #Node, #Lev, and #Conf, respectively. The last column shows the runtimes in second.

The symmetry group orders in Table I range from one trivial symmetry up to approximately 10^{260} symmetries. The largest group order was reported for b19; the largest benchmark in our collection. In our experiments, we observed that the number of symmetries of an n-to-1 multiplexer was reported to be $(log\, n)!$ This number corresponds to all permutations of the multiplexer's control signals.

The largest runtime in Table I is 1843 seconds (reported for b17). Of the 55 total benchmarks, only 3 took more than 1000 seconds to finish. The remaining were solved in less than 400 seconds (including s38584 which has more than a thousand I/Os). All benchmarks with less than a hundred I/Os were processed in less than two seconds. We also observed that b19 was not processed within the time-out limit.

In our experiments, the majority of the benchmarks (76%) did not encounter any conflict. This suggests that our refinement techniques were effective enough to prune away unpromising branches of search. We also assessed the effect of learning by disabling it and re-running our algorithm on all benchmarks that showed conflicts. We observed that, without learning, our algorithm failed to solve 61% of those benchmarks within the time-out limit.

TABLE II. The results of using our symmetry-detection algorithm to solve PP-equivalence checking

Circuit	#Node	#Lev	#Conf	Time (s)	Time (s) from [1]
mux-64	68	13	32	0.29	2.51
mux-128	4144	17	754	46	22
adder-16	141	36	36	0.11	0.05
adder-40	333	84	84	1.02	0.84
b05	161	26	86	0.22	0.19
b12	75	16	30	2.32	$>$ 2000
b14	1057	62	874	12	10
b20	2096	119	1742	190	126
b21	2096	119	1742	210	145
s5378	272730	104	6785	173	1.45
s13207	11201	609	9377	78	$>$ 2000
s15850	4893	232	4200	83	$>$ 2000
s38584	5459	514	3504	188	$>$ 2000
frg2	60	13	24	0.24	0.47
i10	166	30	79	3.25	2.20
rot	206	35	104	0.75	0.4

As part of our study, we encoded several instances of Boolean matching as symmetry detection, and used a slightly modified version of our algorithm to solve them (command `bm2` in ABC). Table II demonstrates the results, and compares the runtimes of our matcher to that proposed in [1] (we chose [1] since it solves large-scale matching and its source code is publicly available). All the results in Table II are averaged over 10 re-runs, where each re-run 1) randomly permuted I/Os of the circuits, and 2) reconstructed the circuits using ABC's synthesis commands to ensure structural difference.

Of the 16 benchmarks in Table II, our Boolean matcher managed to solve all 16 in less than 210 seconds, but the matcher from [1] failed to process 4 within the time-out limit. On the other hand, the matcher from [1] solved one instance (s5378) in less than 2 seconds, but our matcher took 173 seconds to process it. For the remaining benchmarks, both matchers exhibited comparable results.

VI. Conclusions and Future Work

In this paper, we revisited the notion of symmetry in Boolean functions with a special emphasis on group theory. We proposed a new algorithm that searches for symmetries of Boolean functions under permutation of inputs and outputs. We used functional dependency, random simulation and satisfiability to facilitate the search. We also learned from satisfiability counterexamples to avoid similar conflicts. Moreover, we solved instances of Boolean matching by formulating them as symmetry-detection problems. Empirical results confirm the scalability of our algorithm to circuits with hundreds of I/Os. As future research, we plan to extend our algorithm to detect symmetry under permutation and negation of I/Os. Applications of such an algorithm include solving the general Boolean matching problem [2], and reducing samples for logic simulation [16].

References

[1] H. Katebi and I. L. Markov, "Large-scale Boolean matching," in *Advanced Techniques in Logic Synthesis, Optimizations and Applications*, S. P. Khatri and K. Gulati, Eds. Springer, 2011.

[2] G. Agosta, F. Bruschi, G. Pelosi, and D. Sciuto, "A unified approach to canonical form-based Boolean matching," in *DAC*, 2007, pp. 841–846. [Online]. Available: http://doi.acm.org/10.1145/1278480.1278689

[3] C. Scholl, D. Moller, P. Molitor, and R. Drechsler, "BDD minimization using symmetries," *IEEE TCAD*, vol. 18, no. 2, pp. 81–100, Nov. 2006. [Online]. Available: http://dx.doi.org/10.1109/43.743706

[4] K. Chang, I. L. Markov, and V. Bertacco, "Post- placement rewiring by exhaustive search for functional symmetries," *ACM TODAES*, vol. 32, pp. 10–1145, 2007.

TABLE I. THE RESULTS OF APPLYING OUR SYMMETRY-DETECTION ALGORITHM TO AVAILABLE BENCHMARKS

Circuit	#I	#O	|AIG|	#Symms	#Gen	#Node	#Lev	#Conf	Time (s)
mux-4*	6	1	19	2	1	3	2	0	0.01
mux-8	11	1	57	6	2	7	3	0	0.02
mux-16	20	1	158	24	3	13	4	0	0.05
mux-32	37	1	419	120	4	21	5	0	0.10
mux-64	70	1	1085	720	5	31	6	0	0.28
mux-128	135	1	2777	5040	6	43	7	0	1.10
mux-256	264	1	7044	40320	7	57	8	0	5.85
adder-1*	3	2	19	6	2	7	3	0	0.01
adder-16	33	17	144	196608	17	307	18	0	0.19
adder-40	81	41	760	3.298535E12	41	1723	42	0	1.63
b01	6	7	25	2	1	3	2	0	0.01
b02	4	5	20	1	0	1	1	0	0.01
b03	33	34	84	3.185050E7	15	273	17	1	0.15
b04	76	74	443	1.000000E7	0	1	1	0	0.11
b05	34	70	793	1.741824E7	12	211	15	2	0.20
b06	18	15	19	2880	7	57	8	0	0.02
b07	49	57	351	24	3	13	4	0	0.28
b08	29	25	154	1	0	3	2	1	0.03
b09	28	29	84	3.556874E14	16	273	17	0	0.15
b10	27	23	176	1	0	1	1	0	0.02
b11	37	37	610	1	0	1	1	0	0.04
b12	125	127	1002	960	7	57	8	0	1.08
b13	62	63	256	6	2	7	3	0	0.07
b14	276	299	6061	2.652529E32	29	871	30	0	7.08
b15	484	519	8384	2.652529E32	29	871	30	0	267
b17	1451	1512	27514	1.493036E98	90	8191	91	0	1843
b18	3357	3343	71878	9.802584E259	234	54991	235	0	1488
b20	521	512	12186	7.035908E64	58	3423	59	0	73
b21	521	512	12743	7.035908E64	58	3423	59	0	76
b22	766	757	18450	3.732589E97	88	7833	89	0	138
c499	41	32	400	384	4	64	5	37	0.30
c880	60	26	327	16	4	21	5	0	0.06
c5315	178	123	1773	5.662310E8	24	601	25	0	0.67
c7552	207	108	2074	1.460814E19	45	2376325	60	533198	1141
s953	45	52	347	2.585202E22	22	553	24	1	0.22
s1423	91	97	462	2	1	3	2	0	0.19
s5378	214	228	1389	1.431598E22	49	2551	51	1	2.28
s9234	247	250	1958	2.626993E29	50	83070	59	15541	370
s13207	700	790	2719	1.291078E213	294	86731	295	0	56
s15850	611	684	3560	3.759006E87	112	663078	114	4756	165
s38584	1464	1730	12400	8.200341E116	253	198045	255	2415	141
9symml	9	1	211	362880	8	73	9	0	0.12
apex6	135	99	659	2	1	7	3	1	0.07
frg2	143	139	1164	240	5	43	7	1	0.13
i2	201	1	232	2.038573E222	180	42343	184	8985	3.83
i7	199	67	904	3.850825E66	62	3907	63	0	1.54
i10	275	224	1818	1658880	13	183	14	0	1.25
k2	45	45	1998	4	2	13	4	1	0.12
lal	26	19	109	768	5	31	6	0	0.05
pm1	16	13	47	864	7	57	8	0	0.06
rot	135	107	550	1658880	15	241	16	0	0.38
term1	34	10	311	480	6	43	7	0	0.11
x1	51	35	377	8	3	13	4	0	0.04
x2	10	7	54	2	1	3	2	0	0.03
x3	135	99	833	2	1	3	2	0	0.03
x4	94	71	439	7257600	12	157	13	0	0.19

$*$ mux-n is an n-to-1 multiplexer, and adder-n is an n-bit ripple-carry adder with a carry in.

[5] J. S. Zhang, A. Mishchenko, R. Brayton, and M. Chrzanowska-Jeske, "Symmetry detection for large Boolean functions using circuit representation, simulation, and satisfiability," in *DAC*, 2006, pp. 510–515.

[6] V. N. Kravets and K. A. Sakallah, "Generalized symmetries in Boolean functions," in *ICCAD*, 2000, pp. 526–532.

[7] Berkeley Logic Synthesis and Verification Group, "ABC: A system for sequential synthesis and verification. *http://www.eecs.berkeley.edu/ alanmi/abc*."

[8] J. B. Fraleigh, *A First Course in Abstract Algebra*, 6th ed. Addison Wesley Longman, 2000.

[9] A. Mishchenko, S. Chatterjee, and R. Brayton, "FRAIGs: A unifying representation for logic synthesis and verification," UC Berkeley, Tech. Rep., 2005.

[10] A. Mishchenko, S. Chatterjee, and R. Brayton, "Improvements to combinational equivalence checking," in *ICCAD*, 2006, pp. 836–843.

[11] H. Katebi, K. A. Sakallah, and I. L. Markov, "Conflict anticipation in the search for graph automorphisms," in *LPAR*, 2012.

[12] F. Brglez and H. Fujiwara, "A neutral netlist of 10 combinational benchmark circuits and a target translator in fortran," in *ISCAS*, 1985, pp. 663–698.

[13] F. Brglez, D. Bryan, and K. Koiminski, "Combinational profiles of sequential benchmark circuits," in *ISCAS*, 1989, pp. 1929–1934.

[14] S. Yang, "Logic synthesis and optimization benchmarks user guide version 3.0," 1991.

[15] Electronic CAD and Reliability Group at Politecnico di Torino, "ITC'99 benchmarks. *http://www.cad.polito.it/downloads/tools/itc99.html*."

[16] C.-C. Yu, A. Alaghi, and J. P. Hayes, "Scalable sampling methodology for logic simulation: Reduced-ordered monte carlo," in *ICCAD*, 2012.