

Structural Detection of Symmetries in Boolean Functions

Guoqiang Wang¹

Andreas Kuehlmann^{1,2}

Alberto Sangiovanni-Vincentelli¹

¹ University of California at Berkeley, CA, USA

² Cadence Berkeley Labs, Berkeley, CA, USA

Abstract

Functional symmetries provide significant benefits for multiple tasks in synthesis and verification. Many applications require the manual specification of symmetries using special language features such as symmetric data types. Methods for automatically detecting symmetries are based on functional analysis, e.g. using BDDs, or structural methods. The latter search for circuit graph automorphisms which imply functional symmetry. In this paper we present a method for finding symmetries of Boolean functions based on a two-step approach. First, the circuit structure is modified to maximize its structural regularity and thus the number of inherent automorphisms. The next step implements a fast algorithm for detecting the automorphism generators of the circuit graph. The generators provide a compact representation of all automorphisms which in turn encode a subset of the functional symmetries. Because of its pure structural nature, our approach avoids the complexity issues inherent to methods using BDDs, yet it still works automatically and independently from the input specification format. However, the described method may not detect all functional symmetries, however, our experiments demonstrate that it can find the majority of the symmetries present in practical circuits.

1 Introduction

Functional symmetries denote input permutations of Boolean functions that are an invariant with respect to the computed output value. They play an important role in logic synthesis and functional verification as they provide a valuable insight for efficient circuit restructuring and search strategies to check satisfiability and sequential reachability. For example in [1] functional symmetry is exploited to optimize a circuit implementation for low power consumption and delay with limited area overhead. In [2] the authors demonstrate that the size of the Binary Decision Diagram (BDD) of a Boolean function can be reduced significantly if symmetric variables are placed in adjacent positions. Based on this observation a specialized sifting procedure for dynamic variable ordering was developed [3], which plays a crucial role in symbolic model checking. Furthermore, symmetry of input permutations imply a corresponding isomorphism of the function's co-domain and thus can be exploited to restrict the search space for satisfiability. For example, in [4] symmetry breaking clauses are applied for this purpose. Similarly, algorithms for state space traversal, which are key in formal property checking, can significantly benefit from exploiting symmetry of the state-transition relation [5].

A practical approach for detecting symmetry is based on special language features such as symmetric data types [5]. However, they are of little general value as they require the user to know the symmetries and further to actually mark them in the specification. A method for automatically detecting symmetries is preferable as it is beneficial for a broader set of applications and is independent

of the front-end specification method.

The majority of functional approaches for automatically detecting symmetries are based on analyzing multiple co-factors of the function to be tested [3, 6, 7, 8, 9]. All these approaches have the common problem that they require constructing a monolithic BDD for the function which typically demands significant computing resources. Pomeranz, et al. [10] convert the symmetry detection problem for Boolean networks into a test generation problem and then employ Automatic Test Pattern Generation techniques (ATPG) to detect them. Similar to BDD techniques, ATPG methods often suffer from their computational complexity. Furthermore, the method presented in [10] provides only a test and must be applied to a quadratic number of candidates in order to find all symmetries. Chang, et al. [11, 12] introduced the notion of generalized implication supergates and proposed a linear time algorithm for symmetry identification in a multi-netlist. However, similar to the previously mentioned method, their approach is restricted to simple symmetries considering only complementing or non-complementing variable swaps and cannot handle more general symmetries.

There have been multiple approaches for detecting functional symmetries using structural methods. The method described in [4] applies the off-the-shelf graph isomorphism package NAUTY [13] to detect symmetries in CNF formulas. The result is used to generate symmetry breaking clauses to accelerate a following SAT check. However, as reported by the authors, the generic approach is slow and only applicable to CNF representations. In contrast, our method works directly on a NOR circuit representation which is more natural and also applicable to other domains, e.g. logic synthesis.

The NAUTY package is based on the work of McKay [14] and provides a general automorphism detection algorithm that takes a vertex-colored graph as input and returns its automorphism group in form of a set of generators. NAUTY is based on an iterative partition refinement that starts from the initial graph coloring and searches at each refinement step for automorphism generators. Because of its general nature that can handle arbitrary graphs, NAUTY does not perform best in circuit applications. Based on this observation, Manku, et al. [15] proposed a specialized automorphism detection algorithm [16] that is tuned for CTL formulas and BLIF circuit descriptions. The algorithm is based on an interleaved refinement of two simultaneous graph partitioning processes.

Similar to the algorithms of McKay [14] and Manku [16], our algorithm also uses partition refinement to filter possible automorphisms. However, we apply this refinement filter only once as a preprocessing step and then use a brute-force branch-and-bound algorithm to confirm the automorphisms. This approach works efficiently because in our applications the simple filtering step is able to weed out the majority of false candidates; the remaining cases can be quickly confirmed in the second step. Furthermore, before the automorphisms are detected, our approach modifies the circuit

graph to maximize its regularity and thus the potential for finding more symmetries. As shown in the experimental section, the presented method works efficiently for practical circuits and is able to find most of the existing functional symmetries.

2 Preliminaries

2.1 Functional Symmetry

In general, a given function $f(x_1, \dots, x_n)$ is symmetric with respect to a permutation π of its inputs if $f(x_1, \dots, x_n) = f(\pi(x_1, \dots, x_n))$. Clearly, the set of function-invariant input permutations $\Pi = \{\pi_1, \dots, \pi_m\}$ forms a symmetric group under composition. The set $G \subseteq \Pi$ denotes a set of symmetry generators if all elements of Π can be generated by repeated composition of the $g_i \in G$. For example, the function

$$f = x_1x_2 + x_3x_4 + \bar{x}_5x_6 \quad (1)$$

is symmetric with respect to the permutations

$$\begin{aligned} \Pi = \{ & (1, 2, 3, 4, 5, 6), (2, 1, 3, 4, 5, 6), (1, 2, 4, 3, 5, 6), \\ & (2, 1, 4, 3, 5, 6), (3, 4, 1, 2, 5, 6), (4, 3, 1, 2, 5, 6), \\ & (3, 4, 2, 1, 5, 6), (4, 3, 2, 1, 5, 6) \}. \end{aligned}$$

Here, instead of choosing the cycle representation, we use the Cartesian notation for permutations. (k_1, \dots, k_n) denotes the permutation for which the variable x_{k_i} is connected to input i of f . The given set of permutations can be synthesized by repeated application of the following generators

$$G_\Pi = \{(2, 1, 3, 4, 5, 6), (1, 2, 4, 3, 5, 6), (3, 4, 1, 2, 5, 6)\}.$$

As indicated by this example, a small set of generators can encode an exponential number of permutations; thus the concept of symmetry generators provides an efficient mean for compactly expressing a large set of functional symmetries.

A generalized form of symmetry includes input complementation, i.e., a function might be symmetric with respect to a permutation of its inputs combined with complementation of a subset of them. When considering complementation, the following additional generator

$$G'_\Pi = \{(5', 6, 3, 4, 1', 2)\}$$

can be used to describe a significantly larger set of symmetries for the function given in (1). Here the notation (\dots, k'_i, \dots) denotes that variable x_{k_i} is first complemented and then connected to input i of f .

Shannon's original definition of functional symmetry [17] handles the special case for $\pi = (1, \dots, k_i = j, \dots, k_j = i, \dots, n)$, i.e., he defines that a function f is symmetric in x_i and x_j if $f(\dots, x_i, \dots, x_j, \dots) = f(\dots, x_j, \dots, x_i, \dots)$. Shannon's form of individual variable swaps represents a subset of all generators and thus can represent only a subset of the symmetries. For the above given example these symmetries are described by $G_\Pi = \{(2, 1, 3, 4), (1, 2, 4, 3)\}$.

In [10, 18, 19], the concept of functional symmetry is extended to include swaps between groups of inputs. Kravets and Sakallah [9] give another generalization by introducing "higher-order" symmetries based on hierarchical swaps of sets of inputs. The authors of [9] also consider functional symmetry under input complementation. Similar to Shannon's limited definition of symmetry, this concept generally represents a subset of the symmetry generators and thus cannot cover all functional symmetries present in functions. However, for many existing circuits, the hierarchical notation used in [9] covers the majority of all symmetries and thus provides a practical approach.

2.2 Circuit Graph Representation

Most previous work on symmetry detection is based on building BDDs for the function followed by an analysis of the co-factors. These methods are inherently slow and not applicable in many practical cases where BDDs cannot be constructed due to exorbitant memory requirements. Our approach works directly on the circuit representation. As we will show in the experimental section, in all practical cases the algorithm runs very efficiently even for large circuits.

For our analysis, we apply a "semi-canonical" circuit description with a minimum set of base functions in order to maximize the probability to detect internal symmetries. We start our approach from an AND/INV circuit representation of the function as presented in [20] including the described structural simplifications. Figure 1(a) gives an example of a simple AND/INV circuit graph which implements the function $y = \bar{x}_1x_4 + x_2 + \bar{x}_3$. In the drawing the vertices represent AND functions and dots at the arcs indicate complementation of the corresponding function.

Next, the AND/INV structure is converted into a multi-input NOR representation by maximally expanding AND clusters. We further introduce pairs of vertices for each input, representing both polarities of the input function.

More formally, a NOR circuit graph $C = (V, X, X', E)$ is defined as a set of gates V , pairs of inputs X, X' , and a set of directed edges $E \subseteq ((X \cup X' \cup V) \times V)$, where the set of input vertices X and X' represent the positive and negative input functions x_i and \bar{x}_i respectively. Let $y \in V$ denote the primary output of the circuit graph.

The function of output y is computed recursively as:

$$f(v) = \begin{cases} x_v & : \text{if } v \in X \\ \bar{x}_v & : \text{if } v \in X' \\ \sum_{(u,v) \in E} f(u) & : \text{otherwise} \end{cases}$$

For the above example, Figure 1(b) gives the NOR circuit graph representation. Note that the fan-in structure of the shown vertex v has been replicated for the two fan-outs in order to generate NOR gates with a maximum number of inputs.

In this example, the functional symmetry expressed by the generators $G_\Pi = \{(1, 3', 2', 4), (4', 2, 3, 1')\}$ can directly be observed in the circuit structure as graph automorphism. Informally, a graph automorphism is a bijective mapping of the circuit graph vertices that does not alter the graph topology. The idea of our overall algorithm is to detect the set of automorphisms in the circuit graph which directly correspond to functional symmetries. By first rewriting the AND/INV circuit in a locally canonical form [20] followed by a conversion into the described maximal NOR structure, the number of detectable automorphisms is maximized and so the number of symmetries.

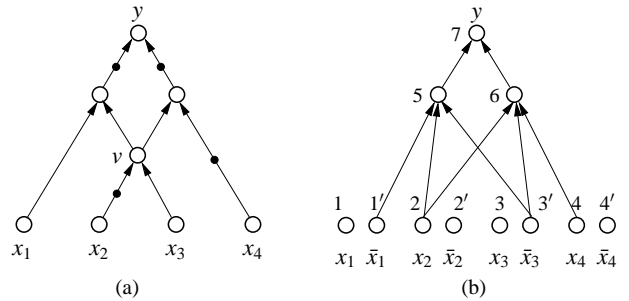


Figure 1: Circuit example for $y = \bar{x}_1x_4 + x_2 + \bar{x}_3$: (a) Two-input AND/INV representation, (b) Multiple-input NOR representation.

2.3 Circuit Graph Automorphism

Our algorithm for structurally detecting functional symmetries is based on computing the existing automorphisms in the circuit graph. A bijective mapping of the graph vertices $a : (X \cup X' \cup V) \rightarrow (X \cup X' \cup V)$ is defined as *circuit graph automorphism* iff:

$$\begin{aligned} &v_1, v_2 \in (X \cup X' \cup V), x_i, x_j \in X, x'_i, x'_j \in X' : \\ &1. \quad e(v_1, v_2) \in E \Leftrightarrow e(a(v_1), a(v_2)) \in E \quad \text{and} \\ &2. \quad x_j = a(x_i) \Leftrightarrow x'_j = a(x'_i) \quad \text{or} \\ &\quad \quad x'_j = a(x_i) \Leftrightarrow x_j = a(x'_i) \end{aligned}$$

Condition 2 ensures that we consider only simultaneous swaps of the input vertex pairs representing the two polarities.

Let $A = \{a_1, \dots, a_k\}$ denote the set of automorphisms of circuit graph C . As for functional symmetries, the set A forms a symmetry group under composition. For the given example of Figure 1(b) the set of automorphisms is

$$A = \{(1, 1', 2, 2', 3, 3', 4, 4', 5, 6, 7), (1, 1', 3', 3, 2', 2, 4, 4', 5, 6, 7), \\ (4', 4, 2, 2', 3, 3', 1', 1, 6, 5, 7), (4', 4, 3', 3, 2', 2, 1', 1, 6, 5, 7)\}.$$

Let $G_A = \{G_{a_1}, \dots, G_{a_k}\}$ denote the set of generators for the symmetry group A . For the example this results in

$$G_A = \{(1, 1', 3', 3, 2', 2, 4, 4', 5, 6, 7), (4', 4, 2, 2', 3, 3', 1', 1, 6, 5, 7)\}.$$

Note that the projection of G_A onto X results in the generators for the functional symmetries, i.e.,

$$G_A \downarrow X = \{(1, 3', 2', 4), (4', 2, 3, 1')\} = G_\Pi.$$

The key approach of our algorithm for symmetry detection is based on first computing the generators for the automorphism followed by a projection onto the input variables.

2.4 Circuit Graph Equivalence Classes

Our algorithm for detecting circuit graph automorphism first generates a set of candidates for vertex swaps followed by a verification step to check individual candidate pairs. The set of candidates are computed by an iterative refinement procedure which produces the coarsest graph partitioning for a structural equivalence relation that is a necessary (but not sufficient) condition for automorphism.

Given the NOR circuit graph $C = (V, X, X', E)$, the equivalence relation $B \subseteq (X \cup X' \cup V) \times (X \cup X' \cup V)$ is defined as follows:

$$u, v, u_1, v_1 \in (X \cup X' \cup V), x_i, x_j \in X, x'_i, x'_j \in X' :$$

1. $(u, u) \in B$
2. $(u, v) \in B \wedge e(u, u_1) \in E \Rightarrow \exists v_1 \in V. (u_1, v_1) \in B \wedge e(v, v_1) \in E$
3. $(u, v) \in B \wedge e(v, v_1) \in E \Rightarrow \exists u_1 \in V. (v_1, u_1) \in B \wedge e(u, u_1) \in E$
4. $(x_i, x_j) \in B \Leftrightarrow (x'_i, x'_j) \in B \quad \text{or} \quad (x_i, x'_j) \in B \Leftrightarrow (x'_i, x_j) \in B$
5. $B(u, v) \Rightarrow |FI(u)| = |FI(v)| \wedge |FO(u)| = |FO(v)|$

$FI(u)$ and $FO(u)$ are the sets of fan-in and fan-out vertices of vertex u , respectively, i.e., $FI(u) = \{v \mid (v, u) \in E\}, FO(u) = \{v \mid (u, v) \in E\}$.

An important property of the above given equivalence relation is a necessary condition for automorphism, i.e.,

$$u = a(v) \Rightarrow (u, v) \in B$$

For the sample circuit given in Figure 1(b) the following vertices are equivalent according to the given definition:

$$B = \{\{7\}, \{5, 6\}, \{1, 4'\}, \{1', 4\}, \{2, 3'\}, \{2', 3\}\}$$

Note that the second automorphism generator for that graph can be computed by swapping the vertices of equivalence classes.

3 Symmetry Generator Detection Algorithm

3.1 Overview of the SG Algorithm

Figure 2 shows the pseudo code of the high-level flow of the symmetry detection algorithm. The algorithm first constructs the AND/INV circuit graph for a Boolean function and applies structural simplifications as described in [21]. The resulting structure is then converted into a NOR circuit graph by forming maximal AND clusters. Next, the coarsest partition according to the above defined equivalence relation is computed. Since this relation is a necessary condition for graph automorphism, the resulting equivalence classes can be used as seeds for checking actual automorphisms.

```

Algorithm SG {
  Construct AND/INV circuit and perform maximal merging;
  Convert to NOR circuit graph;
   $B = \text{Refine\_Partitions}(V, X, X', E)$ ;
  Confirm_Generators( $B$ );
}

```

Figure 2: High-level algorithm for detecting symmetry generators.

Figure 3(a) shows the AND/INV circuit graph for the Boolean function $y = (\bar{x}_1 + x_2)(\bar{x}_3 + x_4)(\bar{x}_8 + \bar{x}_9)(\bar{x}_9 + \bar{x}_{10}) + x_5 + x_6 + \bar{x}_7$, where the two symmetry generators:

$$G_\Pi = \{(1, 2, 3, 4, 6, 5, 7, 8, 9, 10), (1, 2, 3, 4, 5, 6, 7, 10, 9, 8)\}$$

can be easily observed in the given structure. However other functional symmetries, for example the Shannon symmetry between x_5 and x_7 , are hidden by the AND/INV circuit graph because the corresponding AND structure is asymmetrically implemented. A conversion from an AND/INV circuit graph to a maximal NOR circuit graph can retrieve the hidden symmetries as shown in Figure 3(b). Due to the maximal expansion of AND clusters, the resulting NOR circuit graph reveals three additional symmetry generators:

$$G_\Pi = \{(2', 1, 3, 4, 5, 6, 7, 8, 9, 10), (1, 2, 4', 3, 5, 6, 7, 8, 9, 10), \\ (1, 2, 3, 4, 7', 6, 5', 8, 9, 10)\}.$$

3.2 Circuit Graph Partition Algorithm

Figure 4 gives the pseudo-code for computing the coarsest partition B . Our approach is based on the classical partition refinement algorithm given by Hopcroft [22] which is broadly used for many similar purposes. The algorithm uses a greatest fixpoint computation for successively refining a given partition. The initial set of equivalence classes is based on a classification according to inputs and non-inputs and the number of fan-ins and fan-outs. Next, the algorithm iteratively refines the equivalence classes until no further change occurs.

Applying this algorithm to the circuit of Figure 3(b), the initial set of equivalence classes is:

$$B' = \{\{1, 2', 3, 4', 8, 10\}, \{1', 2, 3', 4, 8', 10'\}, \{5, 6, 7'\}, \\ \{5', 6', 7\}, \{9\}, \{9'\}, \{17, 18, 19, 20\}, \{23\}\}.$$

The next iteration of the algorithm will split the equivalence class $\{17, 18, 19, 20\}$ into two parts resulting in:

$$B = \{\{1, 2', 3, 4', 8, 10\}, \{1', 2, 3', 4, 8', 10'\}, \{5, 6, 7'\}, \\ \{5', 6', 7\}, \{9\}, \{9'\}, \{17, 18\}, \{19, 20\}, \{23\}\}.$$

After one more iteration, the fixed point is reached and the final partition result depicted in Figure 3(b) is:

$$B = \{\{1, 2', 3, 4'\}, \{1', 2, 3', 4\}, \{5, 6, 7'\}, \{5', 6', 7\}, \\ \{8, 10\}, \{8', 10'\}, \{9\}, \{9'\}, \{17, 18\}, \{19, 20\}, \{23\}\}.$$

The vertices of each equivalence class are used as candidate pairs for finding automorphisms. The corresponding checking routine is given in the next section.

3.3 The Confirmation Algorithm

The second part of the **SG** algorithm includes the actual confirmation or verification step. Since the equivalence relation computed in the first step does not necessarily imply graph automorphism, the confirmation step checks whether the stronger automorphism condition holds such that all circuit graph vertices can be matched pairwise according to the property given in Section 2.3.

Figures 5 and 6 illustrate the procedures **Confirm_Generators** and **Match**. The algorithm **Confirm_Generators** checks all possible symmetry assignments encoded in the equivalence relation B . We use an index set I to avoid repeated processing of seed symmetry pairs. The index set I is first initialized with all input variable vertices. Then the two loops enumerate all seed symmetry pairs. Each pair is first pushed onto the queue *Process_Queue*. During processing, this queue contains all vertex pairs that still need to be matched for a complete graph automorphism. The list G_A stores the confirmed mapping for a graph automorphism and is initialized with the seed pair. The procedure **Match** is invoked to determine a complete matching of all remaining graph vertices. If an automorphism is detected, the corresponding entries are removed from I to avoid a repeated processing.

Algorithm **Match** performs a systematic attempt to match all remaining vertices by a recursive processing of all vertex pairs of the queue *Process_Queue*. It first makes a decision to pair up two vertices from the corresponding equivalence class and then recursively proceeds to collect more matches and vertex pairs that need to be verified. In case of conflicting matches, a marking mechanism allows to undo all assignments recorded in G_A . To detect whether there is a graph automorphism according to a seed symmetry pair,

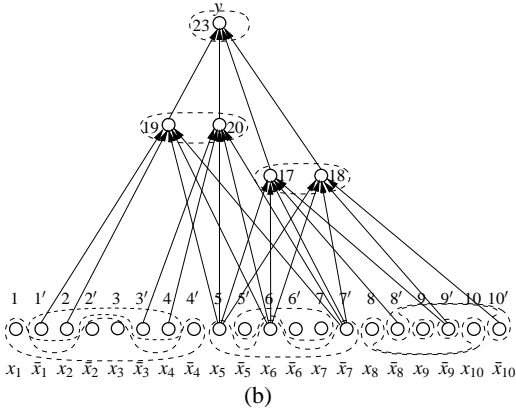
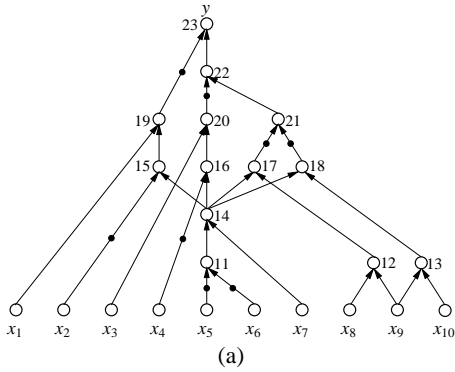


Figure 3: Circuit graph for function $y = (\bar{x}_1 + x_2)(\bar{x}_3 + x_4)(\bar{x}_8 + \bar{x}_9)(\bar{x}_9 + \bar{x}_{10}) + x_5 + x_6 + \bar{x}_7$: (a) Two-input AND/INV representation, (b) Multiple-input NOR representation.

```

Algorithm Refine_Partitions( $(V, X, X', E)$ ) {
   $B' = \{(x_i, x_j), (x'_i, x'_j) \mid x_i, x_j \in X, x'_i, x'_j \in X'\} \cup$ 
     $\{(x'_i, x_j), (x_i, x'_j) \mid x_i, x_j \in X, x'_i, x'_j \in X'\} \cup$ 
     $\{(u, v) \mid u, v \in V : (|FO(u)| = |FO(v)|) \wedge (|FO(u)| = |FO(v)|)\}$ 
  do {
     $B = B'$ ;
     $B' = \{(u, v) \mid (u, v \in (V \cup X \cup X') : (u, v) \in B \wedge$ 
       $\forall e(u, u_1) \in E : \exists v_1. ((u_1, v_1) \in B \wedge e(v, v_1) \in E) \wedge$ 
       $\forall e(v, v_1) \in E : \exists u_1. ((v_1, u_1) \in B \wedge e(u, u_1) \in E)) \wedge$ 
       $(x_i, x_j \in X, x'_i, x'_j \in X' : (x_i, x_j) \in B \Leftrightarrow (x'_i, x'_j) \in B \vee$ 
       $(x_i, x'_j) \in B \Leftrightarrow (x'_i, x_j) \in B)\}$ ;
    } while ( $B' \neq B$ )
  }
  return  $B$ ;
}

```

Figure 4: Partition refinement algorithm for computing the coarsest partition in a circuit graph according to B .

sometimes all possible combinations of the vertices from the same equivalence class must be tried. If all tries fail, the algorithm backtracks to a previous level to undo the decision made before. If the queue *Process_Queue* becomes empty without encountering a conflict, a corresponding graph automorphism is found. The symmetry generator contained in the detected automorphism consists of those variable vertex pairs in G_A . Projecting the graph automorphism onto the input variable domain gives the corresponding symmetry generator.

```

Algorithm Confirm_Generators( $B$ ) {
   $I = X \cup X'$ ;
   $G_\Pi = \phi$ ;
  for ( $\forall x_1 \in I \wedge FO(x_1) \neq 0$ ) {
    for ( $\forall x_2 \in I \wedge (x_1, x_2) \in B$ ) {
       $Process\_Queue = \phi$ ;
       $G_A = \phi$ ;
      Push_On_Process_Queue( $Process\_Queue, (x_1, x_2)$ );
      Put_On_GA( $G_A, (x_1, x_2)$ );
      if (!Match( $Process\_Queue$ )) continue;
      else {
         $G_\Pi = G_\Pi \cup \{G_A \downarrow X\}$ ;
        for ( $\forall (x_3, x_4) \in G_A$ )
          if ( $x_3 \neq x_1$ )  $I = I \setminus \{x_3\}$ ;
          else  $I = I \setminus \{x_4\}$ ;
      }
    }
  }
   $I = I \setminus \{x_1\}$ ;
}
return  $G_\Pi$ ;
}

```

Figure 5: Algorithm for detecting independent automorphism and symmetry generators in a NOR circuit graph.

Figure 7 shows an example for the matching process. To get a valid match of two vertices, both the fan-in and fan-out sides must be checked. In the given example, according the indicated B , the fan-ins and fan-outs of vertices $\{v_1, v_2\}$ have been partitioned into two and three classes, respectively. Clearly, there is only one assignment to match the fan-ins of v_1 and v_2 : x_{11} with x_{22} and x_{12} with x_{21} . However, on the fan-out side, case splits need to be introduced because multiple fan-outs fall into the same equivalence class. It may be required to enumerate all 24 permutations of vertices to get a valid match.

4 Experimental Results and Discussion

The SG algorithm presented in Section 3 has been applied to the LGSynth91 benchmark circuits. A total of 1535 Boolean functions in 54 modules were tested. To make the results comparable with the results in [9], both the number and the average size of hi-

```

Algorithm Match(Process_Queue){
  if(Process_Queue_Is_Empty()) return true;
  (v1, v2) = Pop_Out_Process_Queue(Process_Queue);
  for( $\forall u_1 \in FO(v_1)$ ){
    for( $\forall u_2 \in FO(v_2) \wedge (u_1, u_2) \in B$ ) { /* try all combinations */
      /* check if vertices are already matched */
      if( $\exists u_3 \neq u_2. (u_1, u_3) \in G_A$ ) return false;
      if( $\exists u_3 \neq u_1. (u_3, u_2) \in G_A$ ) return false;
      Mark_Process_Queue(Process_Queue_Mark);
      Mark_GA(GA_Mark);
      Push_On_Process_Queue(Process_Queue, (u1, u2));
      Put_On_GA(GA, (u1, u2));
      if((flag = Match(Process_Queue))) break;
      Undo_Process_Queue(Process_Queue_Mark);
      Undo_GA(GA_Mark);
    }
    /* if last failed then all failed */
    if(!flag) return false;
  }
  .../* handle fan-in side similarly */
  return true;
}

```

Figure 6: Algorithm for matching both fan-in and fan-out sides of a vertex pair.

erarchical symmetry partitions are computed based on the obtained symmetry generators. Table 1 gives the detailed results. The first column represents the design names. Columns 2, 3, and 4 represent the number of primary inputs, primary outputs, and two-input AND gates of the AND/INV graph of a design, respectively. For comparison, the results of the functional approach proposed in [9] are given in Columns 5 and 6. Columns 7 and 8 give the number of symmetry partitions in the circuit and their average size obtained by the SG algorithm. Column 9 shows the run time of the entire SG algorithm and the last column gives the run time for the confirmation step only. The experimental results show that the SG algorithm can detect 3227 independent symmetry partitions, which are approximately 90% of the 3591 partitions detected by the functional method. The structurally obtained symmetry partitions have an average size of 2.71 and the functionally detected ones have an average size of 2.94.

Notice that the structural method finds more symmetry partitions with a relatively small size compared to those obtained by the functional approach. This is because the structural method cannot always use transitivity to combine small symmetry groups into larger ones, e.g., in the presence of XOR graph structures.

The SG algorithm failed to detect existing symmetries in 80 of the 1535 tested Boolean functions, where symmetries can be found by the functional approach. Since the SG approach is based on circuit graph structures, it may fail on some irregularly designed circuits that include functional symmetries. However as pointed out in [9], highly irregular functions are not well designed and other implementations that are more compact and regular might be possible. This suggests that our structural approach to detect symmetries is practically useful.

The experimental results also show the CPU time for each benchmark module. Our experiments were conducted on a Sony VAIO laptop with an Intel Pentium III processor with a clock rate

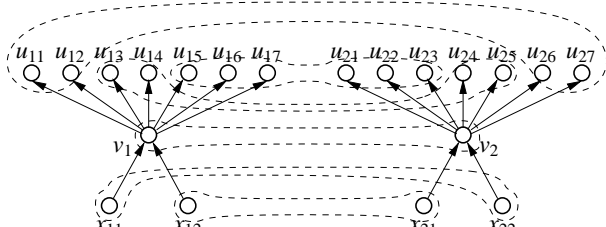


Figure 7: Examples of matches and splits.

of 700MHz. The results demonstrate that the SG algorithm works efficiently. It also shows that the CPU time used for the confirmation step is about one third of the total CPU time of the experiments.

5 Conclusion

In this paper, we presented a simple algorithm for structurally detecting symmetries in Boolean functions. Our approach works directly on an AND/INV circuit graph representation. To detect a large number of automorphisms, a NOR circuit graph is derived. The symmetry detection algorithm first applies circuit graph partition to cluster the vertices into equivalence classes which provide candidates for symmetries. Next, an exhaustive search for vertex mapping is used to detect graph automorphisms. The symmetry generators are then obtained by projecting the automorphism generators onto the input variable domain.

Experiments and a comparison with the results obtained by a functional approach show that the SG algorithm can efficiently detect the majority of symmetry groups in practical Boolean functions. Local rewriting techniques can be applied to further improve the performance of the structural method. The presented algorithm is scalable for large circuits and can be easily generalized for sequential circuits.

6 Acknowledgments

We would like to thank the four anonymous reviewers for their valuable comments. We also want to thank Victor M. Kravets for providing us the detailed symmetry results from his research. One of the authors, Guoqiang Wang, is funded by the Gigascale Systems Research Center.

References

- [1] K. S. Chung and C. L. Liu, "Local transformation techniques for multi-level logic circuits utilizing circuit symmetries for power reduction," in *Proc. of Internat. Symp. on Low Power Electronics and Design*, pp. 215–220, August 1998.
- [2] S.-W. Jeong, T.-S. Kim, and F. Somenzi, "An efficient method for optimal BDD ordering computation," in *Proc. International Conference on VLSI and CAD*, (Taejeon, Korea), November 1993.
- [3] S. Panda, F. Somenzi, and B. F. Plessier, "Symmetry detection and dynamic variable ordering of decision diagrams," in *Proc. International Conference on Computer-Aided Design*, (San Jose, CA), pp. 628–631, November 1994.
- [4] F. A. Aloul, A. Ramani, I. L. Markov, and K. Sakallah, "Solving difficult SAT instances in the presence of symmetry," in *39th Design Automation Conference Proceedings*, (New Orleans, Louisiana), 2002.
- [5] E. A. Emerson and A. P. Sistla, "Symmetry and model checking," in *International Conference on Computer Aided Verification*, (Elounda, Greece), pp. 463–478, Springer-Verlag, 1993.
- [6] C. Scholl, D. Moller, P. Molitor, and R. Drechsler, "BDD minimization using symmetries," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 18, pp. 81–100, February 1999.
- [7] D. Moller, J. Mohnke, and M. Weber, "Detection of symmetry of Boolean functions represented by ROBDDs," in *Proceedings of ICCAD*, (Santa Clara, CA), pp. 680–684, 1993.
- [8] C. Tsai and M. Marek-Sadowska, "Generalized Reed-Muller forms as a tool to detect symmetries," *IEEE Transactions on Computers*, vol. 45, pp. 33–40, January 1996.
- [9] V. N. Kravets and K. A. Sakallah, "Generalized symmetries in Boolean functions," Tech. Rep. (CSE-TR-430-00), The University of Michigan, Ann Arbor, 2000.
- [10] I. Pomeranz and S. M. Reddy, "On determining symmetries in inputs of logic circuits," *IEEE Transactions on CAD of Integrated Circuits and Systems*, vol. 13, pp. 1428–1434, November 1994.
- [11] C.-W. Chang, C.-K. Cheng, P. R. Suaris, and M. Marek-Sadowska, "Fast post-placement rewiring using easily detectable functional symmetries," in *Proceedings of the 37th Design Automation Conference*, pp. 286–289, 2000.
- [12] C.-W. Chang, B. Hu, and M. Marek-Sadowska, "In-place delay constrained power optimization using functional symmetries," in *Design, Automation and Test in Europe*, (Munich, Germany), pp. 377–382, 2001.

| Benchmarks | Number of Primary Inputs | Number of Primary Outputs | Number of two-input And Gates | Functional Approach [9] | | Structural Approach | | | Confirmation (msec) |
|--------------|--------------------------|---------------------------|-------------------------------|-------------------------|----------------------------|----------------------|----------------------------|-----------------|---------------------|
| | | | | Number of Partitions | Average Size of Partitions | Number of Partitions | Average Size of Partitions | CPU time (msec) | |
| 9symml | 9 | 1 | 214 | 1 | 9 | 0 | 0 | 20.3 | 0.0 |
| alu2 | 10 | 6 | 526 | 5 | 2 | 4 | 2 | 47.4 | 0.1 |
| alu4 | 14 | 8 | 990 | 7 | 2.29 | 5 | 2 | 100.3 | 0.1 |
| apex6 | 135 | 99 | 795 | 130 | 2.62 | 134 | 2.43 | 63.1 | 6.4 |
| apex7 | 49 | 37 | 332 | 53 | 3.58 | 55 | 3.33 | 47.0 | 6.6 |
| b1 | 3 | 4 | 18 | 2 | 2 | 2 | 2 | 10.0 | 0.1 |
| b9 | 41 | 21 | 166 | 38 | 2.50 | 35 | 2.31 | 19.0 | 1.0 |
| c8 | 28 | 18 | 323 | 9 | 5 | 13 | 3 | 23.0 | 0.9 |
| cc | 21 | 20 | 99 | 16 | 2.75 | 16 | 2.75 | 15.1 | 0.9 |
| cm138a | 6 | 8 | 129 | 8 | 6 | 8 | 6 | 12.9 | 1.8 |
| cm150a | 21 | 1 | 98 | 3 | 2 | 0 | 0 | 18.7 | 6.7 |
| cm151a | 12 | 2 | 49 | 6 | 2 | 0 | 0 | 14.1 | 2.3 |
| cm162a | 14 | 5 | 66 | 11 | 2.18 | 11 | 2.18 | 12.3 | 0.3 |
| cs163a | 16 | 5 | 66 | 5 | 3.8 | 5 | 3.8 | 12.1 | 0.6 |
| cm42a | 4 | 10 | 23 | 10 | 4 | 10 | 3.2 | 12.6 | 0.6 |
| cm82a | 5 | 3 | 28 | 5 | 2.60 | 5 | 2.40 | 10.6 | 0.4 |
| cm85a | 11 | 3 | 52 | 14 | 2 | 8 | 2 | 12.2 | 0.6 |
| cmb | 16 | 4 | 71 | 4 | 12 | 2 | 12 | 16.2 | 1.9 |
| comp | 32 | 3 | 158 | 49 | 2.29 | 3 | 2 | 39.3 | 15.5 |
| cordic | 23 | 2 | 107 | 14 | 3 | 21 | 2.10 | 20.9 | 5.5 |
| count | 35 | 16 | 163 | 16 | 8.56 | 16 | 8.56 | 35.8 | 11.7 |
| cu | 14 | 11 | 75 | 15 | 3.80 | 17 | 3.35 | 16.1 | 1.9 |
| des | 256 | 245 | 4733 | 685 | 2.37 | 631 | 2.40 | 566.1 | 33.8 |
| dalu | 75 | 16 | 1816 | 48 | 5.40 | 0 | 0 | 448.9 | 12.7 |
| example2 | 85 | 66 | 388 | 104 | 3.60 | 89 | 3.22 | 50.6 | 5.9 |
| f51m | 8 | 8 | 248 | 4 | 2 | 1 | 2 | 16.8 | 0.0 |
| frg2 | 143 | 139 | 2011 | 323 | 3.60 | 342 | 3.18 | 199.8 | 28.8 |
| i1 | 25 | 16 | 63 | 14 | 3.64 | 14 | 3.64 | 15.1 | 1.1 |
| i2 | 201 | 1 | 434 | 8 | 23.50 | 12 | 16 | 2253.4 | 1875.7 |
| i3 | 132 | 6 | 259 | 70 | 2.80 | 70 | 2.80 | 118.5 | 83.1 |
| i4 | 192 | 6 | 434 | 72 | 3.70 | 66 | 2.24 | 29.3 | 1.9 |
| i5 | 133 | 66 | 447 | 66 | 2 | 66 | 2 | 45.8 | 1.0 |
| k2 | 45 | 45 | 2236 | 82 | 3.50 | 62 | 3.56 | 300.8 | 11.5 |
| lal | 26 | 19 | 165 | 43 | 2.67 | 36 | 2.69 | 24.0 | 2.2 |
| majority | 5 | 1 | 20 | 1 | 4 | 1 | 2 | 9.4 | 0.0 |
| mux | 11 | 1 | 39 | 4 | 2 | 1 | 2 | 19.3 | 5.7 |
| my_adder | 33 | 17 | 274 | 152 | 2.11 | 169 | 2 | 44.7 | 6.3 |
| pair | 173 | 137 | 1907 | 644 | 2.94 | 561 | 2.67 | 346.7 | 91.5 |
| parity | 16 | 1 | 62 | 1 | 16 | 15 | 2 | 29.8 | 17.9 |
| pcler8 | 27 | 17 | 105 | 31 | 3.45 | 31 | 3.45 | 54.0 | 2.8 |
| pm1 | 16 | 13 | 77 | 18 | 3.28 | 20 | 2.65 | 16.4 | 0.6 |
| rot | 135 | 107 | 1199 | 257 | 2.44 | 207 | 2.43 | 168.7 | 13.3 |
| sct | 19 | 15 | 160 | 22 | 2.86 | 22 | 2.86 | 21.3 | 0.8 |
| t481 | 16 | 1 | 2133 | 13 | 2 | 0 | 0 | 196.6 | 0.0 |
| term1 | 34 | 10 | 746 | 51 | 2.51 | 36 | 2.19 | 46.5 | 2.2 |
| too_large | 38 | 3 | 8746 | 17 | 2.94 | 5 | 2.60 | 651.2 | 1.8 |
| tt2 | 24 | 21 | 586 | 27 | 3.07 | 31 | 2.35 | 32.7 | 1.5 |
| unreg | 36 | 16 | 149 | 0 | 0 | 0 | 0 | 16.4 | 0.4 |
| vda | 17 | 39 | 1075 | 45 | 2.89 | 31 | 3.13 | 139.5 | 2.6 |
| x1 | 51 | 35 | 1317 | 92 | 2.76 | 83 | 2.63 | 65.7 | 5.2 |
| x2 | 10 | 7 | 67 | 10 | 2.80 | 7 | 2.57 | 12.9 | 0.1 |
| x3 | 135 | 99 | 1464 | 129 | 2.62 | 107 | 2.37 | 75.8 | 5.5 |
| x4 | 94 | 71 | 794 | 129 | 3.09 | 136 | 2.52 | 62.1 | 8.5 |
| z4ml | 7 | 4 | 215 | 8 | 2.50 | 5 | 2 | 14.9 | 0.5 |
| Total | - | 1535 | - | 3591 | 2.94 | 3227 | 2.71 | 6672.7 | 2290.8 |

Table 1: Statistics of symmetry partitions of LGSynth91 MCNC benchmark circuits obtained by the SG algorithm and the functional approach presented in [9].

- [13] B. D. McKay, "nauty user's guide, version 2.2," tech. rep., Australian National University, 2003.
- [14] B. D. McKay, "Practical graph isomorphism," *Congressus Numerantium*, vol. 30, pp. 45–87, 1981.
- [15] G. S. Manku, R. Hojati, and R. K. Brayton, "Structural symmetries and model checking," in *International Conference on Computer-Aided Verification*, (Vancouver, Canada), pp. 159–171, July 1998.
- [16] G. S. Manku, "Structural symmetries and model checking," Master's thesis, University of California, Berkeley, CA, December 1997.
- [17] C. E. Shannon, "A symbolic analysis of relay and switching circuits," *Transactions American Institute of Electrical Engineers*, vol. 57, pp. 713–723, March 1938.
- [18] H. Savoj, *Don't Care in Multi-Level Network Optimization*. PhD thesis, University of California, Berkeley, CA, 1992.
- [19] J. Mohnke, P. Molitor, and S. Malik, "Limits of using signatures for permutation independent Boolean comparison," in *Proceedings of IEEE/ACM Asia and South Pacific Design Automation Conference*, pp. 459–464, IEEE, 1995.
- [20] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai, "Robust Boolean reasoning for equivalence checking and functional property verification," *IEEE Transactions*, vol. 21, pp. 1377–1394, December 2002.
- [21] M. K. Ganai and A. Kuehlmann, "On-the-fly compression of logical circuits," in *International Workshop on Logic Synthesis*, May 2000.
- [22] J. Hopcroft, "An $n \log n$ algorithm for minimizing states in a finite automaton," in *Theory of Machines and Computations* (Z. Kohavi and A. Paz, eds.), (New York), pp. 186–196, Academic Press, 1971.