Symmetry Detection for Large Boolean Functions using Circuit Representation, Simulation, and Satisfiability

Jin S. Zhang[1] Alan Mishchenko[2] Robert Brayton[2] Malgorzata Chrzanowska-Jeske[1]

[1]Department of ECE
Portland State University
Portland, OR 97201
{jinsong, jeske}@ece.pdx.edu

[2]Department of EECS
UC Berkeley
Berkeley, CA 94720
{alanmi, brayton}@eecs.berkeley.edu

ABSTRACT
Classical two-variable symmetries play an important role in many EDA applications, ranging from logic synthesis to formal verification. This paper proposes a complete circuit-based method that makes uses of structural analysis, integrated simulation and Boolean satisfiability for fast and scalable detection of classical symmetries of completely-specified Boolean functions. This is in contrast to previous incomplete circuit-based methods and complete BDD-based methods. Experimental results demonstrate that the proposed method works for large Boolean functions, for which BDDs cannot be constructed.

Categories and Subject Descriptors
B.6.3 [**Logic Design**]: Design Aids – Automatic synthesis.

General Terms
Algorithms, Performance, Experimentation, Theory.

Keywords
Boolean functions, classical symmetries, And-Inverter Graphs, simulation, Boolean satisfiability.

1 INTRODUCTION
A completely specified Boolean function has a non-equivalent *classical symmetry* [7][8] in two variables (a, b) if the function is not changed after swapping a and b:

$$F(\ldots, a, ..., b, \ldots) = F(\ldots, b, ..., a, \ldots).$$

If a function has two pairs of symmetric variables (a, b) and (b, c), it is also symmetric in (a, c). Due to this transitivity of pair-wise symmetries, the symmetric variables can form groups containing more than two variables. The variables in a symmetry group are functionally indistinguishable and can be used interchangeably. This fact is often exploited in circuit optimization and verification. For example, a circuit's placement can be improved by swapping wires corresponding to variables of a symmetry group [5]. In synthesis, a symmetric group of n variables is decomposable using a block with n inputs and $\lceil \log_2(n+1) \rceil$ outputs [15][17].

Other applications of symmetries include Boolean matching [25], BDD minimization [33], and symmetry breaking in Boolean and pseudo-Boolean satisfiability [1][2]. The usefulness of symmetry in formal verification has been extensively investigated [10][13][30].

The methods to detect symmetries are divided into complete ones, which can detect all symmetric variable pairs in a Boolean function, and incomplete ones, which can only detect a subset of them. The previous complete methods used decomposition charts [28], Reed-Muller forms [35], Hadamard transform [32], simulation and ATPG [31], and BDDs [26][20][29][38]. Incomplete methods relied on structural analysis of the circuit [37] and simulation [6]. Of the complete BDD-based methods, the method in [20] is the fastest but requires the construction of BDDs, which is often impossible or takes prohibitive time. This non-robustness is the main limitation of the BDD-based methods for symmetry detection in large functions.

To overcome this limitation of BDDs, several methods to compute symmetries have been proposed that use circuit representations for functions. Some of these approaches are incomplete [37][6]. To our knowledge, the only complete circuit-based approach [31] is based on simulation and ATPG. In this paper, we propose a complete circuit-based symmetry detection algorithm based on structural analysis, simulation, and state-of-the-art in Boolean satisfiability (SAT) [19][27][9]. The proposed algorithm is named CSSS.

First, structural analysis detects classical symmetries for a subset of variable pairs in one sweep over the circuit. It is simpler and faster than the method proposed in [37], which detects not only classical symmetries but also a subset of generalized symmetries [16].

Second, simulation detects non-symmetric variable pairs. Our simulation method is more efficient than simulation in [31][6] because: (a) it targets all rather than individual variable pairs, and (b) in addition to random simulation, it employs guided simulation using distance-1 patterns derived from SAT counter-examples. This simulation scheme is effective for hard variable pairs, substantially reducing the number of SAT calls.

Third, SAT is applied to variable pairs that are unresolved by the previous two techniques. The proposed SAT formulation is based on detecting intermediate equivalences in the circuit during the SAT search [18][21]. The counter-examples discovered by SAT together with distance-1 patterns computed from them are used by the simulation engine again to detect other non-symmetric variable pairs. This tight integration of simulation and SAT is a key factor in the performance improvement of the proposed method.

Furthermore, transitivity analysis of symmetries is applied throughout the process, which can often resolve the symmetry status of a group of variables quickly.

While structural analysis, simulation and SAT are used in previous work for symmetry detection, our key contribution lies in fine-tuning these techniques and synergistically combining them to form a symmetry detection engine applicable to large Boolean functions.

The classical symmetries have been generalized to higher-order symmetries [16] and linear cofactor relationships [40]. Both of these generalizations require efficient detection of the classical symmetries as a special case. Therefore, the current work also contributes to an efficient and scalable detection of these generalized relationships for large Boolean functions.

The paper is organized as follows. Section 2 provides background. Section 3 discusses the overall flow of symmetry computation (Section 3.1) and details our specific contributions to structural analysis (Section 3.2), simulation (Section 3.3), and the use of Boolean satisfiability (Section 3.4). Section 4 shows experimental results. Section 5 concludes and gives directions for future work.

2 BACKGROUND

2.1 Boolean network

A *Boolean network N* is a directed acyclic graph (DAG) such that for each node *i* in *N*, there is an associated representation of a Boolean function f_i and a Boolean variable y_i, where $y_i = f_i$. A node *i* is a fanin of a node *j* if there is a directed edge $\{i, j\}$ and a fanout if there is a directed edge $\{j, i\}$. A node *i* is a transitive fanin (TFI) of a node *j* if there is a directed path from *i* to *j* and a transitive fanout (TFO) if there is a directed path from *j* to *i*. The sources of the graph are the *primary inputs* (PIs) of the network; the sinks are the *primary outputs* (POs). The functionality of a node in terms of its immediate fanins is its *local function*. The functionality of a node in terms of the PIs of the network is its *global function*.

2.2 And-Inv Graphs

In this work, Boolean networks are represented using And-Inv Graphs composed of two-input ANDs and inverters. Structural hashing [18] ensures that ANDs with the same fanins are not duplicated. AIGs compactly represent logic functions and can be easily transformed into multi-input gate graphs to detect symmetries structurally, as was shown in [37]. Simulation of the AIGs can be done efficiently because they are uniform. Finally, AIGs can be combined with an incremental SAT solver [9] for efficient combinational equivalence checking [18][21].

For the details about efficient manipulation of AIGs, refer to [22]. An overview of several other problems solved efficiently by AIG-based simulation and SAT can be found in [24].

2.3 Classical Symmetries

Given a Boolean function $f(X)$ and a subset $S \subseteq X$, S is a symmetric set of f if and only if (iff) f is invariant under any permutation of the subset. If $|S| = 2$, then S is called a symmetric pair of f.

For any pair of variables x_i and x_j in X, there are four cofactors, f_{00}, f_{01}, f_{10} and f_{11}. Variables x_i and x_j have *non-equivalent* (*equivalent*) *symmetry* in $f(X)$ iff $f_{01}=f_{10}$ ($f_{00}=f_{11}$), denoted by NE(x_i, x_j) (E(x_i, x_j)) [7][8]. These two symmetry types are called *classical symmetries*. In the rest of the paper, unless otherwise noted, symmetry refers to classical non-equivalent symmetry.

The following theorems form the basis of transitivity analysis in symmetry detection. Let $f(X)$ be a Boolean function, and let A and B be two disjoint subsets of X. Let variables x_1, x_2 belong to X but not A and B: $x_1, x_2 \in X$, $x_1, x_2 \notin A \cup B$.

Theorem 1. Let variables x_1 and x_2 be symmetric with variables in the sets A and B, respectively. If x_1 and x_2 are symmetric, then all pairs of variables (v_1, v_2) such that $v_1 \in A$, $v_2 \in B$, are symmetric, that is, $A \cup B$ is a symmetry group.

Theorem 2. Let variables x_1 and x_2 be non-symmetric with variables in the sets A and B, respectively. If x_1 and x_2 are symmetric, then all pairs of variables (x_1, v_2) such that $v_2 \in B$, and (x_2, v_1) such that $v_1 \in A$, are non-symmetric.

Theorem 3. Let variables x_1 and x_2 be symmetric with all variables in the sets A and B, respectively. If x_1 and x_2 are non-symmetric, then all pairs of variables (v_1, v_2) such that $v_1 \in A$, $v_2 \in B$, are non-symmetric.

3 SYMMETRY DETECTION ALGORITHM

This section describes the proposed algorithm CSSS for symmetry detection. We start with a high-level overview of the algorithm in Section 3.1 and proceed to discuss the specific aspects: exploiting the circuit structure (Section 3.2) and using simulation (Section 3.3) combined with Boolean satisfiability (Section 3.4). Even though we focus on the computation of non-equivalent symmetry, the proposed algorithm can be easily extended to compute other types of two-variable symmetries [35][40].

3.1 Algorithm overview

The flow of the symmetry detection algorithm is composed of the following steps, as shown in Figure 1.

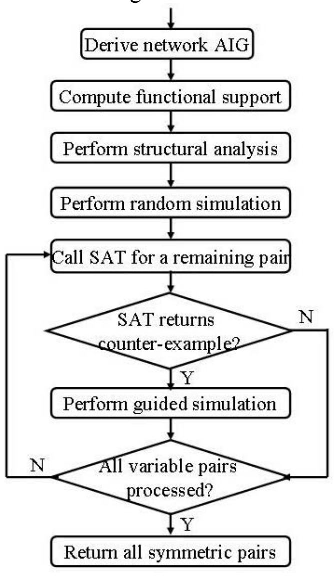

Derive network AIG
Compute functional support
Perform structural analysis
Perform random simulation
Call SAT for a remaining pair
SAT returns counter-example?
N
Y
Perform guided simulation
All variable pairs processed?
N
Y
Return all symmetric pairs

Figure 1. General flow of the symmetry detection algorithm

1. At the beginning, the network is converted into an AIG using structural hashing.
2. The functional supports for the POs are computed. This is necessary since the AIG may contain redundant PI variables belonging to the TFI cones of the POs but having no impact on the PO functionality.
3. A subset of symmetric variable pairs is detected by analyzing the AIG structure. For example, if variables *a* and *b* appear as inputs to only one AND gate in the AIG, or if they appear together as inputs to several AND gates but never appear as inputs independently, then (*a*, *b*) is a symmetric pair. Not all symmetric variable pairs can be detected by examining the AIG structure. For example, all the inputs of a three-input majority gate are symmetric, but the above analysis does not apply to the

AIG derived from the SOP: $ab + bc + ac$. Therefore, symmetry detection based on structural analysis alone is incomplete.

4. Simulation is performed to detect non-symmetric variable pairs. First, random simulation is performed to detect the majority of easy non-symmetric pairs. This is done until saturation, i.e., no new non-symmetric pairs are found during the last n simulation rounds.
5. If some variable pairs are still undecided after simulation, SAT is called to check the symmetry status of the remaining pairs. SAT is performed until all pairs are proved symmetric or a counter-example is found to demonstrate that a variable pair is non-symmetric. In the latter case, the counter-example is passed to the simulation engine. Distance-1 patterns are generated from the counter-example. For example, the distance-1 patterns generated from "0100" are "1100", "0000", "0110" and "0101". Guided simulation is performed using these patterns. This approach is effective for discovering hard-to-detect non-symmetric pairs because the counter-examples reach into narrow functional subspaces. By performing guided simulation from these special points, other difficult pairs can be resolved. Therefore in our algorithm, simulation and SAT are not separate techniques as in the early work, but tightly integrated. This accounts for the reduction of the number of SAT calls, which translates into improved performance for the proposed method.
6. The algorithm finishes when all the variable pairs are proved to be either symmetric or non-symmetric. The set of symmetric variable pairs is returned.

During symmetry detection, each time the symmetry status of a variable pair is determined, transitivity is used to find other symmetries using Theorems 1-3 (Section 2.3). Transitivity analysis is implemented using bit-wise operations on the rows and columns of the bit-matrices storing the symmetry information of the POs.

The following subsections give a detailed description of the three major steps of symmetry detection (Steps 3, 4, and 5). Structural analysis and simulation are applied simultaneously to all POs that have undecided variable pairs, while Boolean satisfiability targets hard-to-prove pairs of one PO.

3.2 Analysis of circuit structure

Symmetry detection based on structural analysis of the circuit is incomplete. Previous work [37] uses graph automorphism to detect both classical and higher-order symmetries. In our structural analysis, we use *implication supergates* (IS) [5]. Instead of performing graph automorphism, we obtain structural symmetries by propagating symmetries through the circuit. Our method is faster and more scalable than [37] because it focuses on the detection of classical symmetries only.

For AIGs, an implication supergate rooted at node n is a multi-input AND gate created by expanding the AND gate rooted at node n until a PI or a complemented edge is reached. Figure 2 shows an example of constructing implication supergates for an AIG.

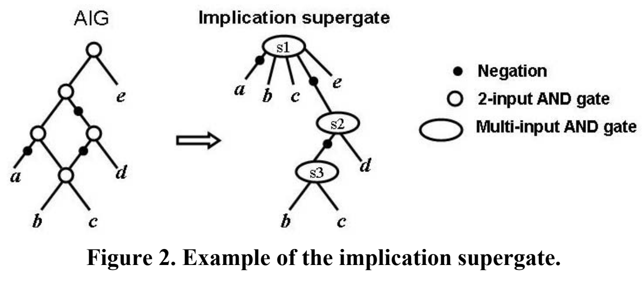


Figure 2. Example of the implication supergate.

Here we describe an efficient implementation of structural symmetry detection by propagating symmetry information of the implication supergates in topological order from the PIs to the POs.

First, the fanins of a supergate are divided into three categories: direct PIs (K_p), complemented PIs (K_n) and all other nodes (K_o). Columns 2, 3, and 4 in Table 1 list the variables in K_p, K_n and K_o for each implication supergate in Figure 2.

Second, candidate symmetries (as shown in Column C-Symm of Table 1) are determined as follows: for classical non-equivalent symmetries, the variables in K_p and in K_n are paired up separately; for classical equivalence symmetry, the variables are paired up by taking one variable from K_p and another from K_n.

Third, the algorithm filters the candidate symmetries by keeping only those for which the following condition holds (as shown in Column K-Symm of Table 1):

- for each fanin in K_o, both variables involved in the symmetry either are symmetric for this fanin or do not belong to its structural support.

Last, the set of resulting symmetries is propagated to its parent supergate if the following conditions hold (as shown in Column P-Symm of Table 1):

- the symmetry holds for at least one fanin in K_o,
- for all fanins in K_o, for which the symmetry does not hold, both variables involved in the symmetry do not belong to the structural support of the fanin.

The symmetric pair after structural analysis for the example in Figure 2 is (b, c).

Table 1. Structural analysis of AIG in Figure 2.

IS	K_p	K_n	K_o	C-Symm	K-Symm	P-Symm
s3	*b*,*c*			(*b*,*c*)	(*b*,*c*)	
s2	*d*		s3			(*b*,*c*)
s1	*b*,*c*,*e*	*a*	s2	(*b*,*c*) (*b*,*e*) (*c*,*e*)	(***b*,*c***)	(*b*,*c*)

3.3 Simulation

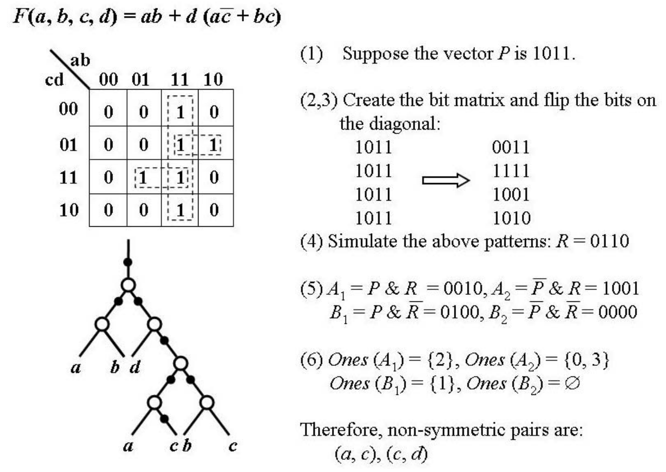
The proposed simulation method detects non-symmetric variable pairs in function f by finding an input pattern such that $f_{01} \neq f_{10}$, where f_{01} and f_{10} are cofactors computed with respect to the considered pair of variables. Previous work [31][6] used a straight-forward simulation scheme that targets each variable pair individually and analyzes the simulation patterns at each output of the circuit, making it slow for large functions.

One advantage of our simulation method is that it simultaneously targets all PI variable pairs appearing in the true support of several POs. In doing so, the manipulation of individual bits is reduced by using bit-parallel computation. The following steps discuss the simulation process for each vector, while Figure 3 illustrates these steps on an example.

1. Get an n-bit Boolean vector P, where n is the number of inputs in the true support of the function. The vector could come from the random vector queue or distance-1 patterns, as discussed in Section 3.1.
2. Create an n x n bit-matrix. Fill n rows with vector P.
3. Flip the bits on the diagonal of the bit-matrix.
4. Treat each row as one simulation pattern and perform bit-parallel simulation of the pattern through the circuit. Let R be the resulting n-bit simulation vector.
5. Create four n-bit vectors: $A_1 = P \,\&\, R$; $A_2 = P \,\&\, \overline{R}$; $B_1 = \overline{P} \,\&\, R$; $B_2 = \overline{P} \,\&\, \overline{R}$, where & is bit-wise AND.
6. Let $A[i]$ denote the i-th value of the bit-vector A, and let $Ones(A)$ denote the set of indexes i, such that $A[i] = 1$. Now

we can mark as non-symmetric all the variable pairs (v_1, v_2), such that $v_1 \in Ones(A_1)$, $v_2 \in Ones(A_2)$, and all the variable pairs (u_1, u_2) such that $u_1 \in Ones(B_1)$, $u_2 \in Ones(B_2)$.

This approach works because after flipping the bits on the diagonal of the bit-matrix in Step 3, every two rows in the resulting matrix represent input vectors needed to compute f_{01} and f_{10} (or f_{00} and f_{11}) for the corresponding input variable pairs. By simulating these vectors and comparing the values of the outputs, we can potentially derive symmetry status for more variable pairs, as shown in the example in Figure 3.



$F(a, b, c, d) = ab + d\,(a\bar{c} + bc)$
ab
cd
00 01 11 10
00 0 0 1 0
01 0 0 1 1
11 0 1 1 0
10 0 0 1 0
a b d
a c b c
(1) Suppose the vector P is 1011.
(2,3) Create the bit matrix and flip the bits on the diagonal:
1011 ⟹ 0011
1011 ⟹ 1111
1011 ⟹ 1001
1011 ⟹ 1010
(4) Simulate the above patterns: $R = 0110$
(5) $A_1 = P \,\&\, R = 0010, A_2 = \bar{P} \,\&\, R = 1001$
$B_1 = P \,\&\, \bar{R} = 0100, B_2 = \bar{P} \,\&\, \bar{R} = 0000$
(6) $Ones\,(A_1) = \{2\}, Ones\,(A_2) = \{0, 3\}$
$Ones\,(B_1) = \{1\}, Ones\,(B_2) = \varnothing$
Therefore, non-symmetric pairs are:
$(a, c), (c, d)$

Figure 3. Example of detecting non-symmetric variable pairs using simulation.

Step 6 above is for classical non-equivalent symmetry. For equivalent symmetry, it should be modified: non-symmetric are all the variable pairs (v_1, v_2), such that $v_1 \in Ones(A_1)$, $v_2 \in Ones(B_2)$, and all the variable pairs (u_1, u_2) such that $u_1 \in Ones(A_2)$, $u_2 \in Ones(B_1)$.

One advantage of the presented simulation technique is that it "implicitly" considers all variable pairs, without having to enumerate through them individually, which makes it fast for large functions. Another advantage is that all operations on the simulation data are performed in bit-parallel fashion, except $O(n)$ operations working with individual bits in Steps 3 and 6.

Variable pairs usually differ in the average number of patterns needed to detect the difference of the cofactors. Some variable pairs cannot be detected using random simulation in reasonable time. This is why we start with random simulation and later switch to guided simulation using counter-examples returned by the SAT solver, together with their distance-1 patterns. The advantage of this simulation model is that it uses the intelligence of the SAT solver to reach into subspaces that are unlikely to be reached using random simulation alone.

For example, random simulation often fails for sparse functions whose Boolean space is populated with many 0's and a few 1's, or vice versa. For such functions, random simulation almost always turns up the more likely value because the probability of getting the unlikely value is very low. Consider a 20-input AND-gate, the probability of getting 1 at the output is equal to $2^{-20} \approx 10^{-6}$, i.e. it takes roughly one million random patterns to distinguish the output from the constant 0 function.

Counter-examples are the patterns producing the unlikely value of a sparse function, thereby distinguishing it from other closely related sparse functions. For practical circuits, the same-valued minterms often lie close and are grouped into cubes. Therefore, the distance-1 patterns of SAT counter-examples target the close neighborhood of a minterm, where the probability of getting the unlikely value is higher than in other parts of the Boolean space. This heuristic does not work for random functions, in which the unlikely values are scattered randomly through the Boolean space.

3.4 Boolean satisfiability

SAT-based symmetry detection is applied to the variable pairs that are not yet proven symmetric or non-symmetric using the analysis of circuit structure and simulation, as discussed above. For an undecided pair, two cofactors are derived and checked for equivalence using a combinational equivalence checker (CEC) [21], which is conceptually similar to [18][11]. Our checker is based on a tight integration of simulation, SAT solving, and fast logic synthesis. For detailed description and experimental evaluations, refer to [21]. The following are the key characteristics of the equivalence checker:

1. Use of fast logic synthesis by way of AIG rewriting [23]. Logic synthesis results in fewer AIG nodes, which leads to faster SAT solving.
2. Use of intelligent simulation by simulating not only distance-1 patterns of SAT counter-examples, but also distance-1 patterns from useful patterns which are successful in distinguishing nodes. This substantially reduces the number of SAT calls, thereby improving the runtime.
3. Use of conflict analysis for better interleaving of the input and output SAT runs. This aids in early detection of intermediate equivalent nodes and speeds up the output proof.
4. Use of CNF-based SAT solver. Recent progress in CNF-based SAT solvers [9] and efficient circuit-to-CNF conversions [36], together with using circuit decision heuristics, increase the speed of SAT solving.

As a result, the employed CEC is faster than earlier approaches [11][12] and leads to improved performance of symmetry detection.

4 EXPERIMENTAL RESULTS

The proposed symmetry detection algorithm CSSS was implemented in ABC [3] as command *print_symm*. The experiments were conducted using a Pentium 4 computer with 1.6GHz CPU and 1Gb RAM with the following goals:

- to analyze the effectiveness of different symmetry detection techniques: structural analysis, simulation, SAT checking and transitivity analysis;
- to demonstrate the efficiency of the proposed method as a result of the tight integration of simulation and SAT;
- to show that CSSS outperforms the BDD-based approaches for large Boolean functions.

In all the experiments, the benchmarks are read into ABC and preprocessed by several iterations of AIG balancing and rewriting to reduce the number of the AIG nodes and logic levels [23]. This preprocessing substantially reduces the runtimes of both CSSS and the BDD-based symmetry computation. Since it is a common step for both computations, the runtime measurements in the tables do not include the preprocessing.

4.1 Analysis of symmetry detection techniques

We divided the MCNC multi-level combinational benchmarks [39] into categories based on the percentage of symmetric variable pairs, as shown in Column 1 of Table 2. The number of benchmarks in each category is shown in Column 2. For example, category "0%" includes 13 benchmarks with no symmetries, whereas category "100%" includes 3 benchmarks, whose outputs are completely symmetric functions. 52 out of 79 MCNC benchmarks contain less than 10% of symmetric variable pairs, as shown in Column 2. Columns 3 to 6 give the average percentages of variable pairs classified using various techniques: structural analysis, simulation, SAT checking, and transitivity analysis.

Table 2. Performance of various symmetry detection techniques on MCNC benchmarks.

Symm. Category	Num. of Benchmark	Structural Analysis	Simulation	SAT	Transitivity Analysis
0%	13	0.00%	99.94%	0.06%	0.00%
0-1%	6	0.26%	99.08%	0.51%	0.15%
1-10%	33	3.88%	94.64%	0.95%	0.57%
10-20%	7	12.77%	84.04%	2.89%	0.30%
20-30%	5	26.09%	73.33%	0.58%	0.00%
30-50%	8	39.71%	54.92%	2.82%	2.54%
50-100%	2	35.80%	44.20%	5.00%	15.00%
100%	3	18.89%	0.00%	12.13%	68.98%

Table 2 confirms that simulation is an important technique in detecting non-symmetric variable pairs. Simulation detected over 94% of non-symmetric variable pairs for the benchmarks with less than 10% of symmetries. As the number of symmetries increases, structural analysis plays a more important role in symmetry detection. Because of the effectiveness of the proposed structural analysis and simulation, SAT is used to process less than 1% of the variable pairs for most MCNC benchmarks. This is desirable because SAT is NP-hard and time-consuming.

4.2 Integration of simulation and SAT

The following experiment compares CSSS with a plain implementation where only random simulation is performed, followed by SAT. The purpose is to demonstrate the effectiveness of the integration of simulation and SAT.

Table 3. Contribution of integrated simulation and SAT.

Name	Total Pairs	Number of SAT Calls			Runtime (s)		
		CSSS	Plain	Ratio	CSSS	Plain	Gain
c1355	26240	105	642	0.16	0.51	2.02	3.96
c1908	11116	98	1583	0.06	0.86	4.68	5.44
c2670	32333	100	11380	0.01	1.78	54.06	30.37
c3540	13579	28	230	0.12	0.51	3.88	7.61
c499	26240	109	642	0.17	0.53	2.19	4.13
c5315	62496	180	3079	0.06	1.56	11.55	7.40
c6288	10792	44	291	0.15	10.44	37.55	3.60
c7552	143390	166	50563	0.00	4.21	300.46	71.37
c880	6436	34	140	0.24	0.14	0.33	2.36
dalu	12540	23	144	0.16	5.36	6.63	1.24
frg2	14523	62	563	0.11	0.38	1.3	3.42
i10	110581	364	14235	0.03	8.25	172.24	20.88
i2	20100	11	62	0.18	0.21	0.48	2.29
i8	9408	3	4	0.75	0.12	0.12	1.00
k2	9361	44	292	0.15	0.44	1.42	3.23
my_adder	3656	9	186	0.05	0.07	0.43	6.14
rot	19429	107	1751	0.06	0.68	5.33	7.84

Experiments show that 31 out of 79 MCNC benchmarks benefit from the simulation and SAT integration. Table 3 lists some relatively large Boolean functions whose number of variable pairs exceeds 5000. Column 1 gives the names of these benchmarks, and Column 2 lists the total number of variable pairs in these functions. Columns 3 (CSSS) and 4 (plain implementation) show the number of SAT calls used to detect non-symmetric variable pairs (only non-symmetric pairs benefit from this integration). Because of this integration, the number of SAT calls is dramatically reduced. This translates into improved runtime, as shown in Column 6 (CSSS), Column 7 (plain implementation) and Column 8 (performance gain is the ratio of Columns 8 and 7). For the largest benchmark, C7552, the SAT call savings are almost 100% with a significant 71x performance improvement.

In summary, tight integration between simulation and SAT makes the proposed method applicable to larger Boolean functions.

4.3 Performance comparison of CSSS vs. BDDs

Table 4 contains the runtime information for the same subset of large MCNC benchmarks, as shown in Table 3, as well as some large benchmarks from ITC [14], ISCAS [4], and PicoJava [34] benchmark suites. The total runtime (Column 5) of our approach includes structural analysis (Column 2), simulation (Column 3), SAT checking (Column 4) and the time it takes to detect the true support of each output. Column 6 ("F-BDD") reports the runtime of the fastest BDD-based symmetry detection algorithm [20] implemented in ABC and tested on the same computer. It includes the time needed to construct the shared BDDs of the PO functions and compute symmetries by traversing the BDDs. Column 7 ("N-BDD") reports the runtime of the naïve BDD-based symmetry computation, which computes the cofactors with respect to all variables pairs and compares them. Column 8 ("Gain-F") gives the performance gain between the proposed method and method in [20], whereas Column 9 ("Gain-N") gives the performance gain between the proposed method and the naïve method. The highlighted entries indicate when the proposed approach performs the same as or better than the other two methods.

Table 4. Runtime comparison.

Name	Proposed CSSS algorithm (s)				F-BDD	N-BDD	Gain-F	Gain-N
	Struct	Sim	SAT	Total	(s)	(s)		
C1355	0.01	0.04	0.05	0.39	2.89	34.69	**7.41**	**88.95**
C1908	0.00	0.00	0.10	0.67	0.94	3.90	**1.40**	**5.82**
C2670	0.00	0.03	0.27	0.89	1.08	16.77	**1.21**	**18.84**
C3540	0.00	0.05	0.11	0.49	3.41	16.59	**6.96**	**33.86**
C499	0.00	0.02	0.02	0.36	3.26	34.43	**9.06**	**95.64**
C5315	0.01	0.12	0.20	1.15	0.91	2.56	0.79	**2.23**
C6288	0.00	0.04	6.85	7.65	xxxx	xxxx	**xxxx**	**xxxx**
C7552	0.00	0.14	0.52	2.15	1.88	17.65	0.87	**8.21**
C880	0.01	0.05	0.04	0.20	0.95	4.24	**4.75**	**21.20**
dalu	0.00	0.03	0.49	3.13	0.55	0.85	0.18	0.27
frg2	0.01	0.09	0.01	0.52	0.38	0.49	0.73	0.94
i10	0.01	0.39	0.75	4.87	6.59	199.30	**1.35**	**40.92**
i2	0.23	0.03	0.01	0.36	0.41	1.46	**1.14**	**4.06**
i8	0.00	0.12	0.00	0.33	0.28	0.44	0.85	**1.33**
k2	0.02	0.08	0.02	0.40	0.36	0.56	0.90	**1.40**
my_ad	0.01	0.01	0.05	0.18	0.10	0.16	0.56	0.89
rot	0.00	0.03	0.04	0.60	0.64	4.83	**1.07**	**8.05**
too_large	0.01	0.02	0.04	0.17	0.30	0.37	**1.76**	**2.18**
b14	9.49	4.08	9.04	54.52	xxxx	xxxx	**xxxx**	**xxxx**
b15	0.31	25.29	64.4	921.2	xxxx	xxxx	**xxxx**	**xxxx**
pj2	0.06	1.03	0.44	9.70	3.23	18.18	0.33	**1.87**
pj3	1.28	12.09	10.24	503.9	xxxx	xxxx	**xxxx**	**xxxx**
s15850	0.05	1.73	9.06	68.24	8.13	53.66	0.12	0.79
s38417	65.48	29.45	4.89	316.6	xxxx	xxxx	**xxxx**	**xxxx**

Table 4 shows that CSSS is faster in 15 out of the 24 cases compared to the fastest BDD-based algorithm [20]. In 5 of these cases BDDs can not be constructed. When CSSS loses, it is mostly within 2x of the best BDD runtime, which is still reasonable. This demonstrates that the proposed method is more robust and can be applied to large functions.

5 CONCLUSIONS AND FUTURE WORK

In this paper, a set of methods is proposed to detect all classical symmetries in completely specified Boolean functions. The methods use the circuit representation and do not require the construction of BDDs, which may be impossible for large designs.

In the proposed approach, a substantial subset of symmetries is detected by a quick traversal of the circuit. Bit-parallel simulation is applied to detect the majority of non-symmetric variable pairs. The simulation is interleaved with a specialized SAT-based equivalence-checking method, which proves or disproves the unresolved variable pairs and provides counter-examples for guided simulation.

The proposed method detects the complete set of symmetries. It is more robust than the efficient BDD-based method and faster even for some of those examples for which BDDs can be efficiently constructed.

This work continues the research started in [24] extending the use of simulation and SAT to EDA applications in logic synthesis. Future work will include developing efficient methods for NPN-equivalence checking of large Boolean functions without using BDDs.

ACKNOWLEDGEMENT

This research was supported in part by SRC contract 1361.001, NSF contract CCR 0312676, and by the California Micro program with our industrial sponsors, Intel, Magma, Synplicity, and Altera. The authors acknowledge Jerry Burch for helpful suggestions.

REFERENCES

[1] F. Aloul, A. Ramani, I. Markov, and K. Sakallah, "Solving difficult instances of Boolean satisfiability in the presence of symmetry", *IEEE Trans. CAD*, Vol. 22(9) , Sep. 2003, pp. 1117-1137.

[2] F. Aloul, A. Ramani, I. Markov, and K. Sakallah, "ShatterPB: Symmetry-breaking for pseudo-Boolean formulas", *Proc. ASP-DAC '04*, pp. 884-887.

[3] Berkeley Logic Synthesis and Verification Group. ABC: A system for sequential synthesis and verification, Release 60303. http://www.eecs.berkeley.edu/~alanmi/abc/

[4] F. Brglez, D. Bryan, and K. Kozminski, "Combinational profiles of sequential benchmark circuits," *Proc. ISCAS '89,* pp. 1929-1934.

[5] C.-W. Chang, C.-K. Cheng, P. Suaris, and M. Marek-Sadowska, "Fast post-placement rewiring using easily detectable functional symmetries", *Proc. DAC '00*, pp. 286-289.

[6] C.-L. Chou, G.-W. Lee, J.-Y. Jou, and C.-Y. Wang, "Graph automorphism-based algorithm for determining symmetric inputs", *Proc. ICCD '04*, pp. 417-419.

[7] D. L. Dietmeyer and P. R. Schneider. "Identification of symmetry, redundancy, and equivalence of Boolean functions". *IEEE Trans. Elec. Computers*, Vol. 16(6), Dec. 1967, pp. 804-817.

[8] C. R. Edward and S. L. Hurst. "A digital synthesis procedure under function symmetries and mapping methods". *IEEE Trans. Computers*, Vol. C-27(11), Nov. 1978, pp. 985-997.

[9] N. Eén and N. Sörensson, "An extensible SAT-solver", *Proc. SAT '03,* pp. 502–518, http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/Main.html

[10] E. A. Emerson and A. P. Sistla, "Symmetry and model checking", *Proc. Formal Methods in System Design '96*, pp. 105-131.

[11] F. Lu, L. Wang, K. Cheng, and R. Huang. "A circuit SAT solver with signal correlation guided learning". *Proc. DATE '03*, pp. 892-897.

[12] F. Lu, L.-C. Wang, K.-T. Cheng, J. Moondanos and Z. Hanna, "A signal correlation guided ATPG solver and its applications for solving difficult industrial cases", *Proc. DAC '03*, pp. 668-673.

[13] C. N. Ip and D. L. Dill, "Better verification through symmetry", *Proc. Formal Methods in System Design '96*, pp. 41-75.

[14] ITC '99 Benchmarks. http://www.cad.polito.it/tools/itc99.html

[15] B.-G. Kim and D. L. Dietmeyer. "Multilevel logic synthesis of symmetric switching functions". *IEEE Trans. CAD*, Vol. 10(4), April 1991, pp. 436-446.

[16] V. N. Kravets and K. A. Sakallah. "Generalized symmetries in Boolean functions". *Proc. ICCAD '00*, pp. 526-532.

[17] V. N. Kravets. *Constructive Multi-Level Synthesis by Way of Functional Properties*. Ph. D. Thesis. University of Michigan, 2001.

[18] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai, "Robust Boolean reasoning for equivalence checking and functional property verification", *IEEE Trans. CAD*, Vol. 21(12), Dec. 2002, pp. 1377-1394.

[19] J. P. Marques-Silva and K. A. Sakallah, "GRASP: A search algorithm for propositional satisfiability", *IEEE Trans. Comp*, Vol. 48(5), May 1999, pp. 506-521.

[20] A. Mishchenko, "Fast computation of symmetries in Boolean functions". *IEEE Trans. CAD*, Vol. 22(11), Nov. 2003, pp. 1588-1593.

[21] A. Mishchenko, S. Chatterjee, R. Brayton, and N. Eén, "Improvements to combinational equivalence checking", *Proc. IWLS '06*. http://www.eecs.berkeley.edu/~alanmi/publications/2006/iwls06_cec.pdf

[22] A. Mishchenko and R. Brayton, "Scalable logic synthesis using a simple circuit structure", *Proc. IWLS '06*. http://www.eecs.berkeley.edu/~alanmi/publications/2006/iwls06_sls.pdf

[23] A. Mishchenko, S. Chatterjee, and R. Brayton, "DAG-aware AIG rewriting: A fresh look at combinational logic synthesis", *Proc. DAC '06.* http//www.eecs.berkeley.edu/~alanmi/publications/2006/dac06_rwr.pdf

[24] A. Mishchenko, J. S. Zhang, S. Sinha, J. R. Burch, R. Brayton, and M. Chrzanowska-Jeske, "Using simulation and satisfiability to compute flexibilities in Boolean networks", *IEEE Trans. CAD*, Vol. 25(5), May 2006, pp. 743-755.

[25] J. Mohnke, P. Molitor, and S. Malik. "Limits of using signatures for permutation independent Boolean comparison". *Formal Methods in System Design*, Vol. 21(2), Sep. 2002, pp. 167-191.

[26] D. Möller, J. Mohnke, and M. Weber. "Detection of symmetry of Boolean functions represented by ROBDDs". *Proc. ICCAD'93*, pp. 680-684.

[27] M. Moskewicz, C. Madigan, Y. Zhao, L.Zhang, and S. Malik. "Chaff: engineering an efficient SAT solver". *Proc. DAC '01*, pp. 530–535.

[28] A. Mukhopadhyay. "Detection of total or partial symmetry of a switching function with the use of decomposition charts". *IEEE Trans. Elec. Computers*. Vol. 16(10), Oct. 1963, pp. 553-557.

[29] S. Panda, F. Somenzi, and B. F. Plessier. "Symmetry detection and dynamic variable ordering of decision diagrams". *Proc. ICCAD '94*, pp. 628-631.

[30] M. Pandey and R.E. Bryant, "Exploiting symmetry when verifying transistor-level circuits by symbolic trajectory evaluation," *IEEE Trans. CAD*, Vol. 18(7), July 1999, pp. 918-935.

[31] I. Pomeranz and S.M. Reddy, "On determining symmetries in inputs of logic circuits", *IEEE Trans. CAD*, Vol. 13(11), Nov. 1994, pp. 1428-1434.

[32] S. Rahardja and B. L. Falkowski. "Symmetry conditions of Boolean functions in complex Hadamard transform", *Electronic letters*, vol. 34, Aug. 1998, pp. 1634-1635.

[33] Ch. Scholl, D. Möller, P. Molitor, and R. Drechsler. "BDD minimization using symmetries". *IEEE Trans. CAD*, Vol. 18(2), Feb. 1999, pp. 81-100.

[34] SUN Microelectronics. *PicoJava Microprocessor Cores*. http://www.sun.com/microelectronics/picoJava/

[35] C.-C. Tsai and M. Marek-Sadowska. "Generalized Reed-Muller forms as a tool to detect symmetries". *IEEE Trans. Computers*, Vol. C-45(1), Jan. 1996, pp. 33-40.

[36] M. N. Velev, "Efficient translation of Boolean formulas to CNF in formal verification of microprocessors", *Proc. ASP-DAC '04*, January 2004, pp. 310-315.

[37] G. Wang, A. Kuehlmann, and A. Sangiovanni-Vincentelli, "Structural detection of symmetries in Boolean functions", *Proc. ICCD '03*, pp. 498-503.

[38] K.-H. Wang and J.-H. Chen. "K-Disjointness paradigm with applications to symmetry detection for incompletely specified functions", *Proc. ASP-DAC*'05, pp. 994-997.

[39] S. Yang. *Logic synthesis and optimization benchmarks*. Version 3.0. Tech. Report. Microelectronics Center of North Carolina, 1991.

[40] J. S. Zhang, M. Chrzanowska-Jeske, A. Mishchenko, and J. R. Burch, "Linear cofactor relationships in Boolean functions", *IEEE Trans. CAD*. To appear in June 2006 issue.