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

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

[1] Portland State University, Portland OR [2] University of California, Berkeley

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 which 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.*

1. Introduction

A completely specified Boolean function has a non-equivalent *classical symmetry* [7][8] in two variables (a, b) if the function does not change 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 the transitivity of pair-wise symmetries, the symmetric variables can form larger 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 [13][15]. Other applications of symmetries include Boolean matching [21], BDD minimization [26], and symmetry breaking in (pseudo-)Boolean satisfiability [1][2]. The usefulness of symmetry in formal verification has been investigated in various works, such as [10][11][27].

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 [24], Reed-Muller forms [32], Hadamard transform [29], simulation and APTG [27], and BDDs [22][29][19][34]. Incomplete methods are based on structural analysis of the circuit [33] and simulation [6]. Of the complete BDD-based methods, the algorithm proposed in [19] 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.

The limitation of BDDs is recognized and several methods to compute symmetries have been proposed, which use the circuit representation for functions present in the designs. Some of these approaches are incomplete [33][6]. To our knowledge, the only complete circuit-based approach is [27], 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 Boolean satisfiability (SAT) [17][23][9].

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 [33], which detects not only classical symmetries but also a subset of generalized symmetries.

Second, simulation detects non-symmetric variable pairs. Our simulation is substantially more efficient than simulation proposed in [27][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 very effective for hard variable pairs, reducing the number of SAT calls needed to disprove them by an order of magnitude or more.

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 [16][20]. 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 [28][14][21] and linear cofactor relationships [36]. Both of these generalizations require efficient detection of the classical symmetries as a special case. Therefore, the proposed 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 describes the background. Section 3 discusses the overall flow of symmetry computation (Section 3.1) and details the specific contributions: the detection of a subset of symmetries by exploiting the circuit structure (Section 3.2), simulation (Section 3.3), and the use of Boolean satisfiability (Section 3.4). Section 4 lists 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*. Its 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 [16][20], composed of two-input ANDs and inverters. For the details about efficient manipulation of AIGs in their functionally reduced form, see [20].

The advantages of AIG representation are:

- The AIGs efficiently compact the representation of logic functions [16] and can be easily transformed into a multi-input AND graph. Structural analysis of this graph yields information about symmetries [33].
- Simulation of the AIGs can be done efficiently because they are uniform and functionally reduced, as described in [20].
- Finally, an incremental SAT solver [9] works efficiently on SAT problems generated from the AIGs.

An overview of several other problems solved efficiently by AIG-based simulation and SAT can be found in [37][41].

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 it 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, 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 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 symmetry [32][36].

3.1 Algorithm overview

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

1. At the beginning, the network is converted into an AIG with structural hashing [16].
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 circuit structure. For example, all the inputs of a three-input majority gate are symmetric, but the above observations do not apply to the AIG derived from the SOP expression: $ab + bc + ac$. 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 after simulating a fixed number of simulation rounds.
5. If some variable pairs are still undecided after simulation, SAT is called on the remaining pairs to check their symmetry status. SAT is performed until all the variable 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 very effective in discovering hard-to-detect non-symmetric pairs because the counter-examples discovered by SAT reach into narrow functional subspaces. By performing guided simulation from these special points, more 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 of 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 applied to derive other symmetries using Theorems 1-3.

The following subsections discuss the three major steps (steps 3, 4, and 5) of symmetry detection in detail.

3.2 Analysis of circuit structure

Symmetry detection based on structural analysis of the circuit is incomplete. Previous work [33] uses graph automorphism to detect classical and higher-order symmetries. Our structural analysis targets only classical symmetries but is much faster. We use *implication supergates* [5][33] to propagate symmetries through the circuit. Our method does not require graph automorphism because it computes only the classical symmetries.

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

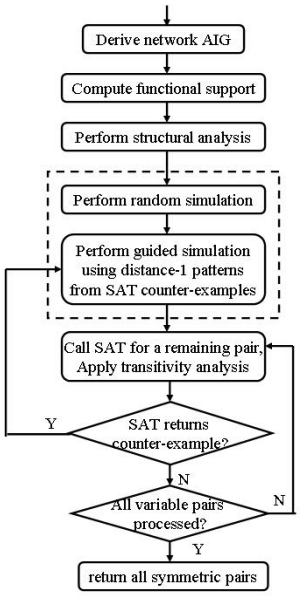
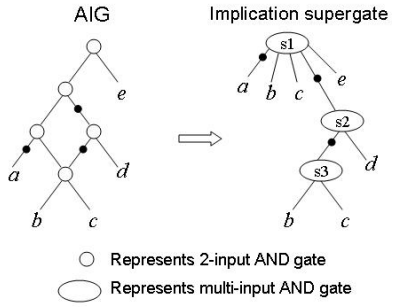


Fig. 1. General flow of the symmetry detection algorithm



Fig, 2. Example of the implication supergate.

Symmetries are detected recursively by visiting the implication supergates in topological order from the PIs to the POs. The fanins of each supergate are separated into three categories: direct PIs (K_p), complemented PIs (K_n) and all other nodes (K_o). For example, for implication supergate $s1$ in Fig. 2, $K_p = \{b, c, e\}$, $K_n = \{a\}$, $K_o = \{s2\}$. Symmetries involving variables in K_p and K_n can be easily determined as follows: for classical non-equivalent symmetries, the variables within K_p and K_n are paired up; for classical equivalence symmetry, we need to pair the variables by taking one variable from K_p and another from K_n. For example, the variable pairs with NE symmetry for $s1$ are $\{b, c\}$, $\{b, e\}$ and $\{c, e\}$; and the variable pairs with E symmetry are $\{a, b\}$, $\{a, c\}$ and $\{a, e\}$.

Then, the algorithm detects symmetries of the other fanins by recursing on each node in K_o. The symmetries of the node are computed by merging the symmetries of its fanins. The resulting set contains a symmetry if both of the following conditions are true: (1) if the symmetry holds for a fanin, the variables do not belong to K_p and K_n on the given level, or belong to the same set, K_p or K_n (for classical non-equivalent symmetry) or to different sets (for classical equivalent symmetry); (2) for each fanin, the symmetry either holds for it, or both variables involved in the symmetry do not belong to the structural support of a fanin.

For example, variable pair $\{b, c\}$ is symmetric for $s3$. Pair $\{b, c\}$ is symmetric for $s2$ because the variables involved in the symmetry do not belong to K_p or K_n of $s2$ (condition (1)) and these variables do not belong to the structural support of other fanins of $s2$ (condition (2)). Similarly it can be shown that $\{b, c\}$ is symmetric for $s1$. Therefore, the final symmetric pairs of $s1$ are: $\{b, c\}$, $\{b, e\}$ and $\{c, e\}$.

It is worth pointing out that even though variable pairs $\{b, e\}$ and $\{c, e\}$ do not appear symmetric in the structure of the original AIG, it is easy to confirm their symmetry after the AIG is expanded into implication supergates. This example illustrates the advantage of using implication supergates to overcome the structural bias of the AIG and reveal additional symmetries, as proposed in [33].

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 [28][6] used a straightforward simulation scheme, which targets individual variable pair 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 a PO. 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 Fig. 3 illustrates these steps using 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 the resulting n-bit simulation vector be R.
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 Fig. 3.

Step 6 above is for classical non-equivalent symmetry. For equivalent symmetry, it should be modified slightly: 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 $u_1 \in Ones(A_2)$, $u_2 \in Ones(B_1)$, are non-symmetric.

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 a fixed number of random simulations 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.

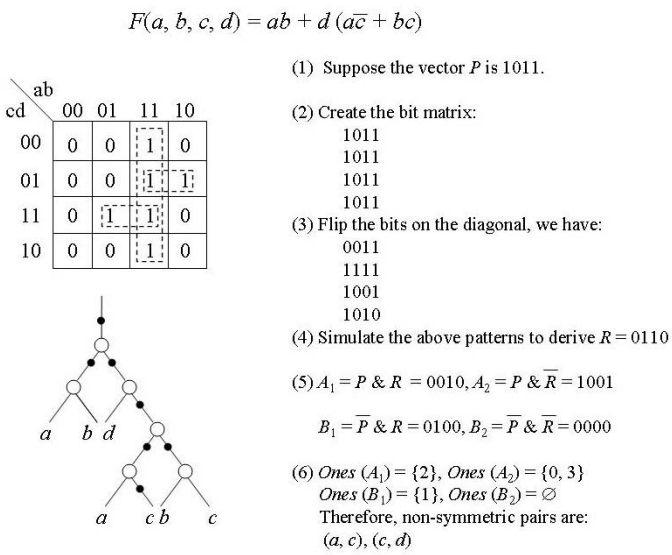


Fig. 3. Example of detecting non-symmetric variable pairs using simulation.

To understand why simulation with distance-1 patterns derived from counter-examples works well in practice, it is helpful to explore the limitations of random simulation. Random simulation fails for some functions because their Boolean space is populated with many 0's and a few 1's, or vice versa. For such "sparse" functions, random simulation almost always turns up the more likely value because the probability of getting the unlikely value is much lower. Consider, for example, 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 function from the constant 0 function.

Counter examples are the ones producing the unlikely value of a sparse function, thereby distinguishing it from other closely related sparse functions. The distance-1 patterns target the close neighborhood of a minterm, in which the function takes the unlikely value. In this neighborhood, we have found empirically that the probability of getting the unlikely value again is higher than in other parts of the Boolean space. The distance-1 heuristic works only for functions appearing in the practical circuits, in which the same-valued minterms often lie close and are grouped into cubes. However, this heuristic would not hold for random functions, in which the unlikely values are scattered randomly through the Boolean space.

3.4 Boolean satisfiability

The SAT-based symmetry checking for a variable pair is essentially a combinational equivalence checking procedure for the two logic cones representing the cofactors with respect to the variables in the given pair. The checking procedure is applied to the variable pairs that are not proven symmetric or non-symmetric using the analysis of circuit structure and simulation, as discussed above.

Each time a variable pair is proved or disproved symmetric using SAT, Theorems 1-3 from Section 2.3 are applied to infer the symmetry status of as many other variable pairs as possible. These variable pairs do not have to be considered again. This noticeably reduces the number of SAT calls.

For each undecided variable pair, two cofactors are computed in the FRAIG manager [20]. Because AIGs in the FRAIG manager are functionally-reduced, the two cofactors are identical iff they are represented by the same node.

In our experiments on symmetry computation, the FRAIG-based equivalence checking is found to be more robust than the straightforward conversion of the logic cones into CNF, followed by the application of a state-of-the-art SAT solvers [23][9] to the miter of these cones. This is because during FRAIGing the internal equivalences in the cofactor logic cones are proved in a topological order, which substantially reduces the computation effort for large problem instances.

Using FRAIG for combinational equivalence checking is less robust than a more general method [16], because the FRAIG package insists on checking the equivalence of all internal candidate cut-points on-the-fly while the method in [16] controls the equivalence checking runtime at the intermediate nodes by a search limit and interleaves these checks with the SAT checks applied to the output miter. However, in our experiments, FRAIGing is always fast, probably because the number of logic levels in the circuits does not exceed 50. The threshold of non-robustness may be reached when processing deeper circuits, for example, multiple time-frames of sequential circuits.

The following observation related to the implementation of the FRAIG package [20] is important to ensure efficient equivalence checking of cofactors. If the FRAIG package contains many intermediate nodes derived by cofactoring in the previous runs, FRAIG construction slows down. This is because the package runs the functional equivalence check for each new node with a potentially large number of nodes not belonging to the graphs of the cofactors under construction. Therefore, in our experiments, we start a new FRAIG manager for each variable pair considered.

4. Experimental results

The proposed symmetry detection algorithm was implemented in ABC [25]. The experiments were conducted using a Pentium 4 computer with 1.6GHz CPU and 1GB RAM. We had three goals in doing these experiments:

1. Analyze the effectiveness of different symmetry detection techniques: structural analysis, simulation, SAT checking and transitivity analysis;
2. Demonstrate the efficiency of the proposed method as a result of the tight integration of simulation and SAT;
3. Prove that S&S-based symmetry detection outperforms the BDD-based approaches for large Boolean functions.

In all the experiments, the benchmarks are read into ABC, followed by a fast pre-processing step. The pre-processing includes several iterations of balancing, refactoring and rewriting the AIG in order to reduce the number of the AIG nodes and logic levels. This dramatically reduces the runtimes for both the S&S and BDD based symmetry computation. Since it is a common step for both computations, the runtimes given in the experiments do not include the pre-processing runtime.

4.1 Contribution of symmetry detection techniques

We separate the MCNC multi-level combinational benchmarks [35] into categories based on the percentage of symmetric variable pairs, as shown in Column 1 of Table 1. The number of benchmarks in each category is shown in Column 2. For example, category "0%" includes 13 benchmarks having 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 1. Performance of various symmetry detection techniques on MCNC benchmarks.

Symm. Category	Num. of examples	Struct	Sim	Sat	Trans
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 1 confirms that simulation is an important technique in detecting non-symmetric variable pairs. It detected over 94% of non-symmetric variable pairs for the benchmarks in the "less than 10%" categories. As the number of symmetries increases, structural analysis plays 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 consumes the longest runtime. The experiment in Section 4.2 shows that the integration of simulation and SAT contributes greatly to the reduction of the needed SAT calls, thereby improving the performance.

4.2 Contribution of the integration of simulation and SAT

The following experiment compares the proposed algorithm with a plain S&S-based symmetry detection scheme where random simulation is performed followed by SAT. The purpose is to demonstrate the effectiveness of the integration of simulation and SAT.

Table 2. Contribution of integrated simulation and SAT.

Name	Total pairs	Number of SAT calls			Runtime (s)		
		our	plain	ratio	our	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

31 out of 79 MCNC benchmarks benefit from the integration. Table 2 shows the comparison for 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 (our proposed algorithm), 4 (plain S&S) and 5 (ratio between Column 3 and 4) reveal information on the number of SAT calls used to detect non-symmetric variable pairs (only non-symmetric pairs benefit from this integration). It is easy to see that because of this integration, the number of SAT calls is dramatically reduced. This directly translates into improved runtime, as shown in Column 6 (our proposed algorithm), Column 7 (plain S&S) and Column 8 (performance gain is the ratio of Column 7 and 8). For the largest function C7552, the improvement is 71 times. We can conclude that the tight integration between simulation and SAT makes the proposed approach applicable to larger Boolean functions.

4.3 Performance comparison of S&S vs. BDDs

Table 3 contains the runtime information for the same subset of large MCNC benchmarks, as shown in Table 2, as well as some large benchmarks from ITC [37], ISCAS [39], and Picojava [40] 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 ("Effic. BDD") reports the runtime of an efficient BDD-based symmetry detection algorithm [19] 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 ("Naïve 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 gives the runtime ratio between the proposed method and [19], whereas Column 9 gives the runtime ratio 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 3. Runtime comparison.

Name	Our approach (s)				Effic.BDD	NaiveBDD	Effic.Ratio	NaiveRatio
	Struct	Sim	Sat	Total				
C1355	0	0.02	0	0.51	5.21	36.47	**0.1**	**0.01**
C1908	0	0.01	0	0.86	0.79	3.24	1.09	**0.27**
C2670	0.01	0.03	0.12	1.78	1.26	7.95	1.41	**0.22**
C3540	0.01	0.03	0	0.51	2.63	15.76	**0.19**	**0.03**
C499	0	0.01	0	0.53	3.12	34.46	**0.17**	**0.02**
C5315	0.01	0.11	0.11	1.56	0.74	2.24	2.11	**0.7**
C6288	0.01	0.1	5.61	10.44	xxxx	xxxx	**xxxx**	**xxxxx**
C7552	0.01	0.16	0.18	4.21	2.5	20.26	1.68	**0.21**
C880	0	0	0	0.14	0.82	5.33	**0.17**	**0.03**
Dalu	0	0.06	0.01	5.36	0.4	0.72	1.33	1.33
frg2	0	0.04	0	0.38	0.21	0.3	1.81	1.27
i10	0.02	0.21	0.72	8.25	8.2	199.3	1.01	**0.04**
i2	0.09	0.03	0	0.21	0.3	1.81	**0.7**	**0.17**
i8	0.01	0.08	0	0.12	0.18	0.28	**0.67**	**0.43**
k2	0.02	0.07	0.01	0.44	0.26	0.49	1.69	**0.9**
my_ad	0	0	0	0.07	0.05	0.08	1.4	**0.88**
rot	0	0.04	0	0.68	0.56	4.5	1.21	**0.15**
b14	8	5.95	6.61	236.2	xxxx	xxxx	**xxxxx**	**xxxxx**
b15	0.33	20.44	26.4	1896	xxxx	xxxx	**xxxxx**	**xxxxx**
pj2	0.06	0.59	0.08	10.42	3.96	13.21	2.63	**0.79**
pj3	1.32	8.45	8.05	107.1	xxxx	xxxx	**xxxxx**	**xxxxx**
s15850	0.05	1.67	0.94	152.6	7.8	50.32	19.57	3.03
s38417	40.47	35.39	1.97	370.1	xxxx	xxxx	**xxxxx**	**xxxxx**

Table 3 shows that the S&S-based method wins 11 out of the 23 benchmarks above, comparing to the efficient BDD-based algorithm proposed in [19], 5 of which whose BDDs can not be built on this

machine. For the benchmarks that S&S 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 combination of these methods makes the most of "positive thinking", "negative thinking" and state-of-the-art in SAT solving to detect the compete set of symmetries. Our results show that the proposed method is much more rugged than the efficient BDD method, and even on those examples where the BDD can be built, the proposed method is faster on 1/3 of the examples.

This work continues the research started in [37][41], which extends 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 BDDs.

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

[4] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE TC*, vol. C-35(8), Aug 1986, pp. 677-691.

[5] C.-W. Chang, C.-K. Cheng, P. Suaris, 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, 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, N. Sörensson, "An extensible SAT-solver", *Proc. SAT '03,* pp. 502–518, http://www.cs.chalmers.se/~een/Satzoo/

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

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

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

[13] 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.

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

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

[16] 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.

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

[18] A. Mishchenko. *EXTRA Library of DD procedures*. http://www.ee.pdx.edu/~alanmi/research/ extra.htm

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

[20] A. Mishchenko, S. Chatterjee, R. Jiang, R. Brayton. "FRAIGs: A unifying representation for logic synthesis and verification". *Technical Report, UC Berkeley, 2004*. http://www.ece.pdx.edu/~alanmi/publications/fraigs08_full.pdf

[21] 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.

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

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

[24] 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.

[25] *ABC: A System for Sequential Synthesis and Verification.* UC Berkeley. http://www-cad.eecs.berkeley.edu/~alanmi/abc/

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

[27] 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.

[28] 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.

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

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

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

[32] 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.

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

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

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

[36] J. S. Zhang, M. Chrzanowska-Jeske, A. Mishchenko, J. R. Burch, "Linear cofactor relationships in Boolean functions", *Trans. CAD*. To appear.

[37] J. S. Zhang, S. Sinha, A. Mishchenko, R. Brayton, M. Chrzanowska-Jeske. "Simulation and satisfiability in logic synthesis", *Proc. IWLS '05*, pp. 161-168.

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

[39] F. Brglez, D. Bryan, and K. Kozminski, "Combinational profiles of sequential benchmark circuits," *Proc. ISCAS '89.*

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

[41] 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", *Trans. CAD*. To appear.