

Review

Symmetry in Boolean Satisfiability

Fadi A. Aloul

American University of Sharjah, P.O. Box 26666, Sharjah, United Arab Emirates;

E-Mail: faloul@aus.edu

Received: 31 December 2009; in revised form: 14 May 2010 / Accepted: 9 June 2010 /

Published: 11 June 2010

Abstract: This paper reviews recent approaches on how to accelerate Boolean Satisfiability (SAT) search by exploiting symmetries in the problem space. SAT search algorithms traverse an exponentially large search space looking for an assignment that satisfies a set of constraints. The presence of symmetries in the search space induces equivalence classes on the set of truth assignments. The goal is to use symmetries to avoid traversing all assignments by constraining the search to visit a few representative assignments in each equivalence class. This can lead to a significant reduction in search runtime without affecting the completeness of the search.

Keywords: boolean satisfiability; symmetries; search

1. Introduction

The last few years have seen a remarkable growth in the use of Boolean Satisfiability (SAT) models and algorithms for solving various problems in Engineering and Computer Science. This is mainly due to the fact that SAT algorithms have seen tremendous improvements in the last few years, allowing larger problem instances to be solved in different application domains. Such applications include formal verification [1], FPGA routing [2], power estimation [3], fault tolerance [4], network assignments [5], wireless communications [6], and scheduling [7]. SAT has also been extended to a variety of applications in Artificial Intelligence, including other well-known NP-complete problems.

SAT solvers have traditionally been used to solve *decision* problems. Given a set of Boolean variables and constraints expressed in products-of-sum form (also known as conjunctive normal form (CNF)), the goal is to identify a variable assignment that will satisfy all constraints in the problem or prove that no such assignment exists. It is well known that the SAT problem is NP-complete [8] and that any algorithmic approach for solving it will require at least worst-case exponential time in its size.

Most powerful SAT solvers today are search based and use intelligent techniques to explore new regions of the search space while looking for a satisfying assignment. These intelligent techniques helped extend the application of SAT solvers to large problem instances consisting of thousands of variables and millions of constraints. Despite these advances, the tremendous growth in today's designs is continuously outpacing the capabilities of existing SAT solvers. Many SAT instances remain hard to solve due to the significant size and complexity of the underlying systems they represent.

Prasad *et al.* observed that Boolean functions arising in Engineering applications are “structured” [9]. Unlike random problems, structured problems have non-uniform distributions and consist of clusters of variables. In this paper, we summarize the main findings in exploiting and analyzing the SAT instance's structure to improve the search process. Specifically, we review an approach to accelerate SAT search by exploiting symmetries in the instance. The approach consists of an automated flow for (1) finding all syntactic symmetries of CNF formulas and (2) utilizing the symmetries to add symmetry-breaking predicates that effectively prune the search space and accelerate SAT solvers. Presented experimental results show the advantage of symmetry breaking for search algorithms.

The remainder of this paper is organized as follows. Section 2 presents an overview of Boolean Satisfiability. Section 3 describes basic symmetry definitions and notations. In Section 4, graph automorphism, which is used to identify symmetries, is described. Symmetry detection and breaking are discussed in Sections 5 and 6, respectively. Experimental results are presented in Section 7. Finally, the paper is concluded in Section 8.

2. Boolean Satisfiability

Recent years have seen significant advances in Boolean Satisfiability (SAT) solving. These advances have led to the successful deployment of SAT solvers in a wide range of problems in Engineering and Computer Science. The SAT problem involves finding an assignment to a set of binary variables that satisfies a given set of constraints or proving that no such assignment exists. In general, these constraints are expressed in *products-of-sum* form, also known as *conjunctive normal form* (CNF). SAT constraints will be referred to as *clauses* in the paper. A CNF formula, φ on n binary variables, $x_1, \dots, x_n$, consists of the conjunction (AND) of m clauses, $\omega_1, \dots, \omega_m$, each of which consists of the disjunction (OR) of literals. A *literal* is an occurrence of a Boolean variable or its complement.

As an example, the CNF instance:

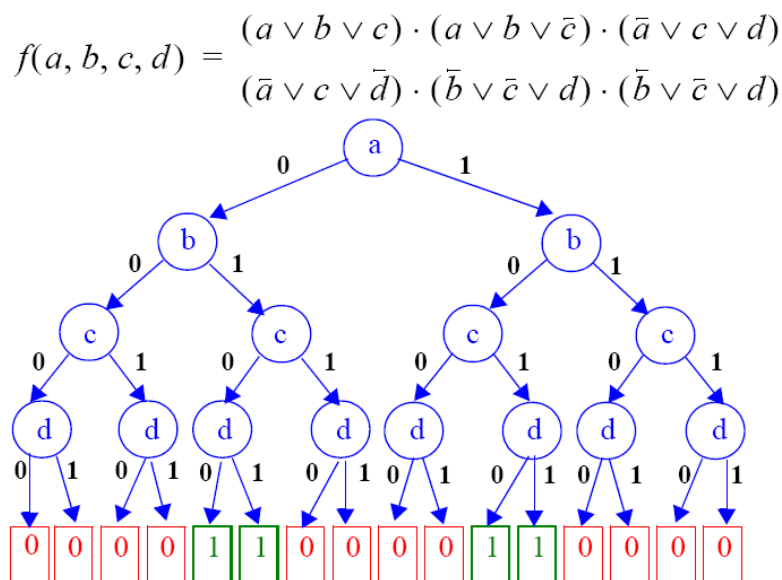
$$f(a, b, c) = (a \vee b) \cdot (b \vee c) \quad (1)$$

consists of three variables, two clauses, and four literals. The assignment $\{a = 1, b = 0, c = 0\}$ leads to a conflict, whereas the assignment $\{a = 1, b = 0, c = 1\}$ satisfies f . Note that a problem with n variables will have 2^n possible assignments to test. The above example with three variables has eight possible assignments. An instance with 100 variables will have $1.27e + 30$ assignments. Assuming a processor that can verify an assignment every one nanosecond, the processor will complete testing all 2^{100} assignments in $4e + 12$ years.

Despite the SAT problem being NP-Complete [8], there have been dramatic improvements in SAT solver technology over the past decade. This has led to the development of several powerful SAT algorithms that are capable of solving problems consisting of thousands of variables and millions of constraints. Such solvers include Grasp [10], zChaff [11], Berkmin [12], MiniSAT [13].

Most powerful SAT solvers are based on the original Davis-Putnam-Logemann-Loveland (DPLL) backtrack search algorithm [14]. The algorithm performs a depth first search process that traverses the space of 2^n variable assignments until a satisfying assignment is found (the formula is *satisfiable*), or all combinations have been exhausted (the formula is *unsatisfiable*). The search process proceeds as follows. Originally, all variables are unassigned. The algorithm begins by choosing a decision assignment to an unassigned variable. A decision tree is maintained to keep track of variable assignments. An example of a decision tree is shown in Figure 1. After each decision, the algorithm determines the implications of the assignment on other variables. This is obtained by forcing the assignment of the variable representing an unassigned literal in an unresolved clause, whose all other literals are assigned to 0, to satisfy the clause. This is referred to as the *unit clause* rule. If no conflict is detected, the algorithm makes a new decision on a new unassigned variable. Otherwise, the backtracking process un-assigns one or more recently assigned variables and the search continues in another area of the search space.

Figure 1. An example of a satisfiable SAT instance showing its corresponding decision tree.



SAT solvers have been extended with several powerful algorithms to further expedite the search process. One of the best algorithms is known as the conflict analysis procedure [10] and has been implemented in almost all SAT solvers. Whenever a conflict is detected, the procedure identifies the causes of the conflict and augments the clause database with additional clauses, known as *conflict-induced clauses*, to avoid regenerating the same conflict in future parts of the search process. In essence, the procedure performs a form of learning from the encountered conflicts. Significant

speedups have been achieved with the addition of conflict-induced clauses, as they tend to effectively prune the search space.

Intelligent decision heuristics and random restarts [11] also played an important role in enhancing the SAT solvers performance. The developers of the state-of-the-art SAT solver, zChaff [11], proposed an effective decision heuristic, known as VSIDS, and implemented several other enhancements, including random restarts, which lead to dramatic performance gains on many CNF instances.

Over the years, SAT has been directly applied to various electronic design automation (EDA) computational tasks. In general, the overhead of reducing many of these problems to SAT is small. Below, we briefly survey a number of EDA applications to which SAT has been successfully introduced.

One of the important applications of SAT is circuit verification. Due to the increasing complexity of modern hardware designs, verifying the correctness of these designs is becoming increasingly difficult. SAT has been successfully applied to formal verification, which has attracted much interest from industry, since unlike simulation, which only reveals the presence of bugs, formal verification can prove their absence. Specifically, combinational equivalence checking has been solved using SAT [15], and later, several attempts have been made to solve sequential equivalence checking problems using SAT [16]. Additionally, SAT has been used for bounded model checking [1], microprocessor verification [17], and functional vector generation [18]. Besides verification, SAT has also been heavily applied in automatic test pattern generation (ATPG) [19] and extended to delay fault testing [20]. SAT has also been used in FPGA routing [2], global routing [21], logic synthesis [22], cross talk noise analysis [23], power estimation [3], and power leakage estimation [24].

Additionally, SAT has been extended to a variety of applications in Artificial Intelligence, including other well known NP-complete problems such as graph colorability, vertex cover, hamiltonian path, and independent sets [25].

3. Symmetry Definitions and Notations

The symmetry of a discrete object is a permutation of its components that leaves the object intact. For example, the rotations of a spatial solid, e.g., a cylinder, that leave its shape unchanged or the negation of a Boolean variable in a Boolean formula, e.g., x in $(x + y) \cdot (\bar{x} + y) \cdot (y + z)$, that does not affect the formula or the function it represents.

Permutational symmetries can be classified as either *syntactic* or *semantic* symmetries. Given a Boolean function, a semantic symmetry is a permutation of variables that does not change the *value* of the function under any variable assignment. On the other hand, a syntactic symmetry, also known as a structural symmetry, is a permutation that does not change the *representation* of the function. Hence, a syntactic symmetry is also a semantic symmetry. In this paper, we focus on using syntactic symmetries to improve SAT search.

In this paper, we will use the *cycle notation* to express the permutations. For example, a permutation that swaps elements 2 and 3, and maps 4 to 5, 5 to 6, and 6 to 4, while mapping all other elements to themselves will be represented as $\phi = (23)(456)$.

In general, the number of permutations in a permutation group can be exponentially large. Rather than explicitly representing the set of permutations in a group, they are typically represented implicitly

using *irredundant sets of generators*. A set of generators is a set of group elements such that any other group element can be obtained from the products of the generators.

As an illustration, consider the permutations $\varphi_1 = (12)$ and $\varphi_2 = (23)$ expressed in cycle notation. Taking the composition, *i.e.*, product, of $\varphi_1 \cdot \varphi_2$ generates the permutation $\varphi_3 = (123)$, which is the result of performing φ_2 followed by φ_1 .

A set of generators is irredundant if it is not possible to express any of its elements as a product of other elements. An interesting observation from group theory is that an irredundant set of generators for a group with $N > 1$ elements contains at most $\log_2 N$ elements [26]. For example, the group of *all* permutations on k elements has $k!$ permutations, but can be efficiently generated by only two generators: $\varphi_1 = (12)$ and $\varphi_2 = (12\dots k)$. Thus, the use of irredundant sets of generators to express the complete set of permutations ensures *exponential* compression.

In this paper, we will assume a total ordering $x_1 < x_2 < \dots < x_n$ of the variables $x_1, x_2, \dots, x_n$ and consider the induced lexicographic ordering of the 2^n truth assignments, *i.e.*, 0–1 strings of length n . Given an equivalence partition on these assignments, the *lex-leader* of a particular equivalence class, *i.e.*, orbit, is defined as its lexicographically smallest element. Adding a *lex-leader predicate* ensures that a Boolean function evaluates to true only on lex-leaders of orbits.

For example, consider a Boolean function with four variables. Assume an orbit consists of the assignments $\{0100, 1000, 0010, 0001\}$. The lex-leader of the given orbit is $\{0001\}$. Adding a lex-leader predicate will falsify the first three assignments.

In terms of a CNF formula, a permutation of literals is a symmetry of a given CNF formula if Boolean consistency is observed and the formula is preserved under the permutation. In other words, every clause must map into a clause with the same polarities of literals. The addition of a *symmetry-breaking predicate* (SBP), expressed as a set of CNF clauses, allows only one of many equivalent variable assignments to be a potential solution to the formula. An SBP is a *full* SBP if it selects exactly a single element from each orbit; otherwise it is known as a *partial* SBP. A *lex-leader* SBP (LL-SBP) is an SBP that selects lex-leaders only and is also classified as a full SBP. It is essential for partial SBPs, however, to select lex-leaders among other elements. Such SBPs are known as *partial lex-leader* SBPs (PLL-SBPs). Note that the addition of an SBP to the original CNF formula does not affect its satisfiability, but restricts the possible solutions to those selected by the SBP. In other words, if the original formula is satisfiable, the number of solutions may decrease considerably after pre-processing. However, if the original instance is unsatisfiable, the “number of equivalent paths leading nowhere” will be reduced and the SAT solver is likely to conclude faster that no solution exists.

As an example, assume the CNF formula:

$$f(a, b, c, d, e, f, g) = (\bar{c} \vee e \vee f)(a \vee c)(b \vee c)(\bar{d} \vee e)(d \vee f) \quad (2)$$

The group of all permutations of the CNF formula consists of four permutations $\{E, (ab), (d\bar{d})(ef), (ab)(d\bar{d})(ef)\}$ that can be efficiently obtained from the products of the two generators $\varphi_1 = (ab)$ and $\varphi_2 = (d\bar{d})(ef)$. Note that E is the identity symmetry. Permuting the literals of the formula according to all four permutations yields the same formula. This is shown in Table 1.

Table 1. Permuting the literals of the CNF formula in (2) according to all its symmetries.

Permutation	CNF Formula
E	$(\bar{c} \vee e \vee f) \cdot (a \vee c) \cdot (b \vee c) \cdot (\bar{d} \vee e) \cdot (d \vee f)$
(ab)	$(\bar{c} \vee e \vee f) \cdot (b \vee c) \cdot (a \vee c) \cdot (\bar{d} \vee e) \cdot (d \vee f)$
$(dd)(ef)$	$(\bar{c} \vee f \vee e) \cdot (a \vee c) \cdot (b \vee c) \cdot (d \vee f) \cdot (\bar{d} \vee e)$
$(ab)(dd)(ef)$	$(\bar{c} \vee f \vee e) \cdot (b \vee c) \cdot (a \vee c) \cdot (d \vee f) \cdot (\bar{d} \vee e)$

4. Graph Automorphism

The problem of identifying all symmetries is known as the graph automorphism problem [27]. Given a graph, a symmetry, *i.e.*, an automorphism, is a permutation of its vertices that maps edges to edges. In the case of directed graphs, edge orientations must be maintained. In terms of complexity, there are no known worst-case polynomial-time algorithms for solving the graph automorphism problem. Nevertheless, the problem is rarely difficult in practice and is not believed to be NP-complete nor to be in P. Algorithms typically finish in linear-time if the symmetry is trivial, and in polynomial-time for graphs of bounded vertex degree [28].

The complexity of solving the graph automorphism problem can be further simplified by the use of vertex labels. Each vertex is labeled by a color or an integer and the goal of the problem is reduced to mapping vertices to vertices of the same label.

In this paper, we will use the powerful graph automorphism tool, Saucy [29].

5. Detecting Symmetries in CNF Formulas

In this section, we describe how to identify symmetries in CNF formulas using graph automorphism [27]. The main idea is to express a CNF formula as an undirected graph such that the symmetry group of the graph is isomorphic to the symmetry group of the CNF formula. The first attempt to identify such symmetries by modeling CNF formulas as graphs was proposed in [30], and subsequently refined by Crawford *et al.* [31]. The construction proceeds as follows. Assuming a CNF formula with V variables and C clauses, of which C_2 are binary clauses and C_x are clauses of size three or more (single literal clauses are removed by preprocessing the CNF formula), a graph is constructed as follows:

- A single vertex is created for each clause in C_x .
- Two vertices are created for each variable, representing its positive and negative literals.
- Edges are added connecting a clause vertex to its respective literal vertices.

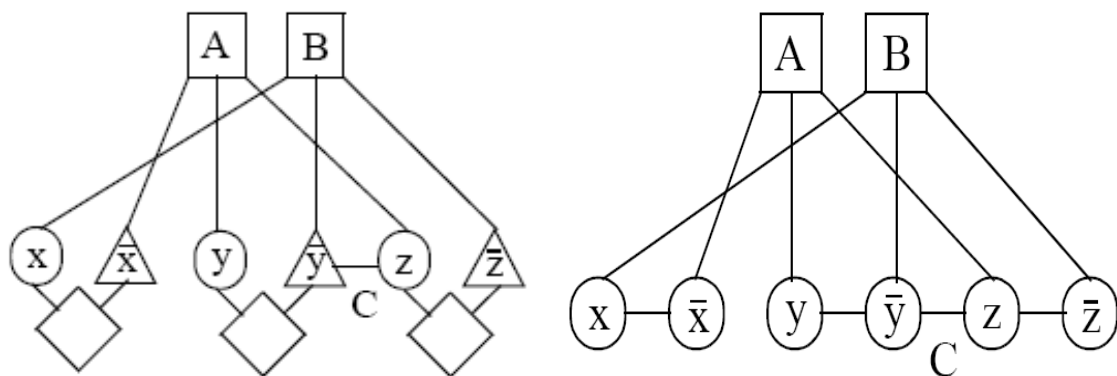
Vertices corresponding to clauses, positive literals, and negative literals are colored differently. Boolean consistency is ensured by adding an extra vertex for each variable and two edges connecting the new vertex to the literals of that variable. The new set of vertices is assigned a new label. Clauses in C_2 are represented by adding a single edge between the two literals of the clause. This construction yields a total of $3V + C_x$ vertices with four unique colors of vertices. An example of the construction is shown in Figure 2(a). Note that the presented construction cannot detect phase shift symmetries, *i.e.*, symmetries of the form $x \rightarrow \bar{x}$.

The construction in [31] was further improved in [32] to handle phase-shift symmetries and their compositions with permutational symmetries. The construction in [32] also produces a smaller graph compared to previous methods. Assuming a CNF formula with V variables and C clauses, of which C_2 are binary clauses and C_x are clauses of size three or more (single literal clauses are removed by preprocessing the CNF formula), a graph is constructed as follows:

- A single vertex is created for each clause in C_x .
- Two vertices are created for each variable, representing its positive and negative literals.
- Edges are added connecting a clause vertex to its respective literal vertices.

Vertices corresponding to clauses and variables are colored differently. Boolean consistency is ensured by adding an edge between vertices of opposite literals. Clauses in C_2 are represented by adding a single edge between the two literals of the clause. The construction yields a total of $2V + C_x$ vertices with two unique colors of vertices. An example of the construction is shown in Figure 2(b).

Figure 2. Conversion of the CNF formula $(\bar{x} \vee y \vee z) \cdot (x \vee \bar{y} \vee \bar{z}) \cdot (\bar{y} \vee z)$ to a graph for symmetry extraction purposes using the constructions (a) in [31] and (b) in [32]. Different shapes correspond to different vertex colors.



6. Breaking Symmetries in CNF Formulas

Once the symmetries are identified, the next step is to break them in the CNF formula. This is accomplished by adding symmetry breaking predicates (SBPs) that choose lex-leaders (LL), *i.e.*, lexicographically smallest assignments, in each orbit or equivalence class.

Crawford *et al.* [31] laid the theoretical foundation for constructing SBPs for CNF formulas that possess permutational symmetries. Their construction assumes a given variable ordering and consists of two nested conjunctions:

- An outer conjunction over all permutations in the group of symmetries.
- An inner conjunction over all variables in the permutation.

This results in the selection of just the lex-leaders from each orbit and breaks all symmetries. Given a group of symmetries $\Pi = \{\pi_1, \dots, \pi_m\}$ for a CNF formula defined over a set of totally-ordered variables $x_1 < x_2 < \dots < x_n$, the LL-SBP is defined as follows:

$$PP(\pi) = \bigcap_{1 \leq i \leq n} [\bigcap_{i \leq j \leq i-1} (x_j = x_j^\pi)] \rightarrow (x_i \leq x_i^\pi) \quad (3)$$

$$LL-SBP(\Pi) = \bigcap_{1 \leq i \leq m} PP(x_i) \quad (4)$$

We will refer to $PP(\pi)$ as the *permutation predicate* for permutation π . By introducing n “equality” variables, $e_j \equiv (x_j = x_j^\pi)$, each PP can be translated to a CNF formula with $5n$ clauses and $0.5n^2 + 13.5n$ literals. This is derived as follows. For each PP, we introduce variables $e_1, \dots, e_n$. For each e_j variable, we add four clauses, each consisting of three literals, in order to define the equality relationship with $(x_j = x_j^\pi)$. Furthermore, the outer conjunction over all n variables in (3) produces n clauses of increasing sizes; the first, second, and n -th clause have a size of two, three, and $n+1$ literals, respectively. Hence, all n clauses have a total of $(n^2 + 3n)/2$ literals. An example of this construction is shown in Table 2.

Table 2. Example showing the creation of SBPs for the single-cycle permutation $(abcd)$ using the formula in (3).

SBP	Clauses
$(a \rightarrow b) \cdot$	$(\bar{a} + b) \cdot$
$((e_1) \rightarrow (b \rightarrow c)) \cdot$	$(\bar{e}_1 + \bar{b} + c) \cdot$
$((e_1 e_2) \rightarrow (c \rightarrow d)) \cdot$	$(\bar{e}_1 + \bar{e}_2 + \bar{c} + d) \cdot$
$((e_1 e_2 e_3) \rightarrow (d \rightarrow a)) \cdot$	$(\bar{e}_1 + \bar{e}_2 + \bar{e}_3 + \bar{d} + a) \cdot$
$(e_1 = (a = b)) \cdot$	$(e_1 + a + b)(e_1 + \bar{a} + \bar{b})(\bar{e}_1 + a + \bar{b})(\bar{e}_1 + \bar{a} + b) \cdot$
$(e_2 = (b = c)) \cdot$	$(e_2 + b + c)(e_2 + \bar{b} + \bar{c})(\bar{e}_2 + b + \bar{c})(\bar{e}_2 + \bar{b} + c) \cdot$
$(e_3 = (c = d))$	$(e_3 + c + d)(e_3 + \bar{c} + \bar{d})(\bar{e}_3 + c + \bar{d})(\bar{e}_3 + \bar{c} + d)$

This approach breaks *all* permutational symmetries and will be referred to as *full* symmetry breaking. The downside is that the number of symmetries in the symmetry group is usually exponential in the number of problem variables, making such full symmetry breaking impractical. This is addressed in [31] by building a *symmetry tree* and pruning it to remove unnecessary duplication. This, unfortunately, still does not preclude the need to consider an exponential number of permutations in order to break all symmetries.

For symmetry breaking to be effective in practice, the computational overhead of generating and manipulating the SBPs must be significantly less than the runtime savings they yield due to search pruning. Since breaking all symmetries can lead to an exponentially large SBP, which will significantly slow down the SAT solver, the authors in [33] suggested *partial* symmetry breaking. The idea is to break a few symmetries whose SBPs produce short CNF clauses. This would provide better runtime and memory trade-offs. The partial symmetry breaking ideas include creating permutation predicates for a subset, instead of all, generators. *Irredundant generators* are good candidates for

symmetries to be broken because they cannot be expressed in terms of each other. Another idea is to create bit predicates for the first k , instead of all bits in a permutation.

As an example, the following formula:

$$f(a,b,c,d) = (a \vee b \vee c \vee \bar{d}) \cdot (a \vee b \vee \bar{c} \vee d) \cdot (a \vee \bar{b} \vee c \vee d) \cdot (a \vee b \vee c \vee d) \quad (5)$$

has a total of 11 permutations. Two of the permutations represent the irredundant set of generators, namely: $\varphi_1 = (ab)(cd)$ and $\varphi_2 = (abc)$. Producing the SBPs for the two generators is more efficient than generating the SBPs for all 11 permutations, yet should still be able to lead to significant savings in SAT search runtime.

7. Experimental Results

In this section, we empirically show the advantage of breaking symmetries in CNF formulas. The experiments were performed on an Intel Xeon 3 Ghz machine with 4 GB of RAM running Linux. The runtime limit for all experiments was set to 1000 seconds. The benchmarks included the pigeon-hole (*hole*) [34], global routing (*s3*) [21], FPGA routing (*fpga* and *chnl*) [32], randomized Urquhart (*urq*) [35], and xor-chains (*xor*) [36]. We used the SAT solvers MiniSAT [13] and RSat [37] to test the advantage of SBPs. MiniSAT and RSat won the gold medals in the industrial category in the SAT 2005 and SAT 2007 competitions [38], respectively. Saucy [29] was used to detect the symmetries and only the irredundant generators were used to generate the SBPs. The tool Shatter [32] was used to generate the SBPs.

Table 3 shows the number of symmetries and generators identified in each instance. The numbers clearly show the significant savings obtained when generators are used to represent the complete set of symmetries. Furthermore, it is clear that all instances contain an exponential number of symmetries.

Table 3. Symmetry statistics of the tested instances.

Instance	S/U	# of Symmetries	# of Generators
chnl10_11	UNS	4.19631E + 28	39
chnl10_12	UNS	6.04269E + 30	41
chnl10_13	UNS	1.02121E + 33	43
chnl11_12	UNS	7.31165E + 32	43
chnl11_13	UNS	1.23567E + 35	45
chnl11_14	UNS	2.42191E + 37	47
fpga12_10_sat	SAT	5.41777E + 16	28
fpga12_11_sat	SAT	1.78786E + 18	29
fpga12_9_sat	SAT	5.41777E + 14	25
fpga13_10_sat	SAT	1.89622E + 17	28
fpga13_11_sat	SAT	1.2515E + 19	30
fpga13_12_sat	SAT	9.01083E + 20	32
hole10	UNS	1.4485E + 14	19
hole11	UNS	1.91202E + 16	21

Table 3. *cont.*

hole12	UNS	2.98275E + 18	23
hole13	UNS	5.42861E + 20	25
hole8	UNS	14631321600	15
hole9	UNS	1.31682E + 12	17
s3-3-3-10	SAT	34828517376	28
s3-3-3-1	SAT	8707129344	26
s3-3-3-3	SAT	69657034752	29
s3-3-3-4	SAT	26121388032	27
s3-3-3-8	SAT	34828517376	28
Urq3_5	UNS	536870912	29
Urq4_5	UNS	8.79609E + 12	43
Urq5_5	UNS	4.72237E + 21	72
x1_16	UNS	131072	17
x1_24	UNS	16777216	24
x1_32	UNS	4294967296	32
x1_36	UNS	68719476736	36
TOTAL		2.43444E + 37	941

Table 4 compares the MiniSAT and RSat solvers' runtimes for the original CNF instances and the instances augmented with SBPs. The table also shows the time needed by Saucy to detect all symmetries and the instance's result as satisfiable (SAT) or unsatisfiable (UNS). Several observations are in order: (1) the symmetry detection runtime is negligible in most cases, (2) the symmetry detection runtime and SAT search runtime are not correlated, and (3) the addition of SBPs for the generators only significantly reduces the SAT search runtime in most cases. The only exception is some of the satisfiable global routing instances which are easily solved by the SAT solvers in their original case, nevertheless, the SAT solvers are also able to easily solve the instances with the augmented SBPs. Similar results using different solvers and machines were obtained in [32,33].

Table 4. SAT solver runtimes for the original instances and the instances augmented with SBPs. Symmetry detection runtimes are also provided. All runtimes are in seconds.

Instance	S/U	Saucy Time (Sec)	RSAT Time (Sec)			MiniSAT Time (Sec)		
			Orig	w/SBP	Speedup	Orig	w/SBP	Speedup
chnl10_11	UNS	0.04	>1000	0.01	>100000	67.92	0.01	6792
chnl10_12	UNS	0.05	>1000	0.01	>100000	115.26	0.01	11526
chnl10_13	UNS	0.07	>1000	0.01	>100000	108.97	0.01	10897
chnl11_12	UNS	0.07	>1000	0.01	>100000	520.44	0.01	52044
chnl11_13	UNS	0.08	>1000	0.01	>100000	>1000	0.01	>100000
chnl11_14	UNS	0.09	>1000	0.01	>100000	>1000	0.01	>100000
fpga12_10_sat	SAT	0.04	0.01	0.001	10	0.01	0.01	1
fpga12_11_sat	SAT	0.05	0.01	0.001	10	>1000	0.01	>100000
fpga12_9_sat	SAT	0.03	0.04	0.01	4	0.01	0.01	1

Table 4. cont.

fpga13_10_sat	SAT	0.05	0.05	0.001	50	0.01	0.01	1
fpga13_11_sat	SAT	0.06	0.08	0.001	80	0.01	0.01	1
fpga13_12_sat	SAT	0.07	0.01	0.01	1	0.01	0.01	1
hole10	UNS	0.01	348.65	0.001	348650	65.85	0.01	6585.0
hole11	UNS	0.01	>1000	0.001	>1000000	>1000	0.01	>100000
hole12	UNS	0.01	>1000	0.001	>1000000	>1000	0.01	>100000
hole13	UNS	0.03	>1000	0.001	>1000000	>1000	0.01	>100000
hole8	UNS	0	0.18	0.001	180	0.49	0.01	49.0
hole9	UNS	0	51.14	0.001	51140	3.24	0.01	324.0
s3-3-3-10	SAT	0.42	2.25	0.09	25	0.69	0.26	2.7
s3-3-3-1	SAT	0.24	0.08	0.75	0.1	1.44	0.12	12
s3-3-3-3	SAT	0.4	0.07	0.16	0.4	0.31	0.19	1.6
s3-3-3-4	SAT	0.37	0.79	0.36	2.2	0.21	0.14	1.5
s3-3-3-8	SAT	0.35	0.4	0.27	1.5	0.16	0.17	0.9
Urq3_5	UNS	0.03	369.57	0.1	3695	84.44	0.19	444.4
Urq4_5	UNS	0.08	>1000	135.54	>7.4	>1000	27.25	36.7
Urq5_5	UNS	0.37	>1000	>1000	1	>1000	780.4	1.3
x1_16	UNS	0	0.01	0.01	1	0.02	0.001	20
x1_24	UNS	0.01	34.04	0.01	3404	2.77	0.03	92.3
x1_32	UNS	0.01	1.33	0.01	133	13.84	0.01	1384
x1_36	UNS	0.03	188.77	0.29	650.9	42.85	0.16	267.8
TOTAL		3	11997	1138		9029	809	

8. Conclusions

Recent algorithmic advances in Boolean Satisfiability (SAT) techniques, along with highly efficient solver implementations, have enabled successful applications of SAT technology to a wide range of applications, and particularly in electronic design automation (EDA). Nevertheless, many SAT instances remain hard to solve due to the significant size and complexity of the underlying systems that they represent. This paper reviews recent approaches that analyze the SAT instance's structure and exploit the structure to improve the search process. Specifically, it describes an automated flow that detects syntactic symmetries in CNF formulas and utilizes them to add symmetry-breaking predicates that effectively prune the search space and accelerate SAT solvers. Strong empirical evidence is presented that shows that symmetry breaking can yield significant speed-ups for a variety of benchmark classes.

References

1. Biere, A.; Cimatti, A.; Clarke, E.; Fujita, M.; Zhu, Y. Symbolic model checking using SAT procedures instead of BDDs. In Proceedings of the Design Automation Conference (DAC), New Orleans, LA, USA, June 1999; pp. 317–320.
2. Nam, G.; Aloul, F.; Sakallah, K.; Rutenbar, R.A. Comparative study of two boolean formulations of FPGA detailed routing constraints. *IEEE Trans. Comput.* **2004**, *53*, 688–696.

3. Sagahyroon, A.; Aloul, F. Using SAT-based techniques in power estimation. *Microelectronics J.* **2007**, *38*, 706–715.
4. Aloul, F.; Kandasamy, N. Sensor Deployment for Failure Diagnosis in Networked Aerial Robots: A Satisfiability-Based Approach. In *Lecture Notes on Computer Science*; Marques-Silva, J., Sakallah, K.A., Eds.; Springer-Verlag: Berlin, Germany, 2007; Volume 4501, pp. 369–376.
5. Aloul, F.; Al-Rawi, B.; Aboelaze, M. Routing in optical and non-optical networks using boolean satisfiability. *J. Commun.* **2007**, *2*, 49–56.
6. Aloul, F.; Tarhuni, M. PN Code acquisition using boolean satisfiability techniques. In Proceedings of the IEEE Wireless Communications & Networking Conference, Budapest, Hungary, 5–8 April 2009; pp. 632–637.
7. Aloul, F.; Al-Rawi, B.; Al-Farra, A.; Al-Roh, B. Solving employee timetabling problems using boolean satisfiability. In Proceedings of the IEEE Innovations in Information Technology Conference, Dubai, UAE, November 2006; pp. 1–5.
8. Cook, S. The Complexity of Theorem Proving Procedures. In Proceedings of the Annual ACM Symposium on the Theory of Computing, Shaker Heights, OH, USA, 1971; pp. 151–158.
9. Prasad, M.; Chong, P.; Keutzer, K. Why is ATPG easy? In Proceedings of the Design Automation Conference (DAC), New Orleans, LA, USA, June 1999; pp. 22–28.
10. Marques-Silva, J.; Sakallah, K. GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Trans. Comput.* **1999**, *48*, 506–521.
11. Moskewicz, M.; Madigan, C.; Zhao, Y.; Zhang, L.; Malik, S. Chaff: Engineering an Efficient SAT Solver. In Proceedings of the Design Automation Conference (DAC), Las Vegas, NV, USA, June 2001; pp. 530–535.
12. Goldberg, E.; Novikov, Y. BerkMin: A Fast and Robust SAT-solver. In Proceedings of the Design Automation and Test Conference in Europe (DATE), Paris, France, March 2002; pp. 142–149.
13. Een, N.; Sorensson, N. An Extensible SAT-solver. In Proceedings of the International Conference on Theory and Applications of Satisfiability Testing (SAT), Santa Margherita Ligure, Italy, May 2003; pp. 502–508.
14. Davis, M.; Longman, G.; Loveland, D. A Machine Program for Theorem Proving. *J. ACM* **1962**, *5*, 394–397.
15. Marques-Silva, J.; Glass, T. Combinational Equivalence Checking Using Satisfiability and Recursive Learning. In Proceedings of the Design Automation and Test Conference in Europe (DATE), Munich, Germany, March 1999; pp. 145–149.
16. Bjesse, P.; Claessen, K. SAT-based Verification without State Space Traversal. In Proceedings of the Formal Methods in Computer-Aided Design (FMCAD), Austin, TX, USA, November 2000; pp. 372–389.
17. Mneimneh, M.; Aloul, F.; Weaver, C.; Chatterjee, S.; Sakallah, K.; Austin, T. Scalable Hybrid Verification of Complex Microprocessors. In Proceedings of the Design Automation Conference (DAC), Las Vegas, NV, USA, June 2001; pp. 41–46.
18. Fallah, F.; Devadas, S.; Keutzer, K. Functional Vector Generation for HDL Models Using Linear Programming and 3-Satisfiability. In Proceedings of the Design Automation Conference (DAC), San Francisco, CA, USA, June 1998; pp. 528–533.

19. Marques-Silva, J.; Sakallah, K. Robust Search Algorithms for Test Pattern Generation. In Proceedings of the International Symposium on Fault Tolerant Computing (FTCS), Seattle, WA, USA, June 1997; pp. 151–161.
20. Chen, C.; Gupta, S. A Satisfiability-Based Test Generator for Path Delay Faults in Combinational Circuits. In Proceedings of the Design Automation Conference (DAC), Las Vegas, NV, USA, June 1996; pp. 209–214.
21. Aloul, F.; Ramani A.; Markov, I. L.; Sakallah, K. Generic ILP *versus* Specialized 0-1 ILP. In Proceedings of the International Conference on Computer-Aided Design (ICCAD), San Jose, CA, USA, November 2002; pp. 450–457.
22. Memik, S.; Fallah, F. Accelerated Boolean Satisfiability-Based Scheduling of Control Data Flow Graphs for High-Level Synthesis. In Proceedings of the International Conference on Computer Design (ICCD), Freiberg, Germany, September 2002; pp. 395–401.
23. Chen, P.; Keutzer, K. Towards True Crosstalk Noise Analysis. In Proceedings of the International Conference on Computer-Aided Design (ICCAD), San Jose, CA, USA, November 1999; pp. 132–138.
24. Aloul, F.; Hassoun, S.; Sakallah, K.; Blaauw, D. Robust SAT-Based Search Algorithm for Leakage Power Reduction. In *Lecture Notes on Computer Science*; Youm, H.Y., Yung, M., Eds; Springer-Verlag: Berlin, Germany 2002; Volume 2451, pp. 167–177.
25. Creignou, N.; Kanna, S.; Sudan, M. *Complexity Classifications of Boolean Constraint Satisfaction Problems*; Society for Industrial Mathematics: Philadelphia, PA, USA, 2001.
26. Hungerford, T. Algebra. In *Graduate Texts in Mathematics*; Springer: New York, NY, USA; 1973.
27. McKay, B.; Practical Graph Isomorphism. *Congressus Numerantium* **1981**, 30, 45–87.
28. Babai, L. Automorphism Groups, Isomorphism, Reconstruction. In *Handbook of Combinatorics*; MIT Press: Cambridge, MA, USA, 1996; Volume 2, Chapter 27, pp. 1447–1541.
29. Darga, P.; Sakallah, K.; Markov, I. Faster Symmetry Discovery using Sparsity of Symmetries. In Proceedings of the Design Automation Conference (DAC), Anaheim, CA, USA, June 2008; pp. 149–154.
30. Crawford, J. A Theoretical Analysis of Reasoning by Symmetry in First-Order Logic. In Proceedings of the AAAI Workshop on Tractable Reasoning, San Jose, CA, USA, July 1992; pp. 17–22.
31. Crawford, J.; Ginsberg, M.; Luks, E.; Roy, A. Symmetry-Breaking Predicates for Search Problems. In Proceedings of the International Conference Principles of Knowledge Representation and Reasoning, Cambridge, MA, USA, November 1996; pp. 148–159.
32. Aloul, F.; Ramani, A.; Markov, I.L.; Sakallah, K. Solving Difficult Instances of Boolean Satisfiability in the Presence of Symmetries. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* **2003**, 22, 1117–1137.
33. Aloul, F.; Sakallah, K.; Markov, I.L. Efficient Symmetry-Breaking for Boolean Satisfiability. *IEEE Trans. Comput.* **2006**, 55, 549–558.
34. DIMACS Challenge Benchmarks. Available online: <ftp://Dimacs.rutgers.EDU/pub/challenge/sat/benchmarks/cnf>. (Accessed on 11 June 2010).
35. Urquhart, A. Hard Examples for Resolution. *J. ACM* **1987**, 34, 209–219.

36. SAT 2002 Competition. Available online: <http://www.satlive.org/SATCompetition/submittedbenchs.html>. (Accessed on 11 June 2010).
37. Pipatsrisawat, K; Darwiche, A. A New Clause Learning Scheme for Efficient Unsatisfiability Proofs. In Proceedings of the Conference on Artificial Intelligence, Chicago, IL, USA, July 2008; pp. 1481–1484.
38. The International SAT Competition. Available online: <http://www.satcompetition.org/> (Accessed on 11 June 2010).

© 2010 by the authors; licensee MDPI, Basel, Switzerland. This article is an Open Access article distributed under the terms and conditions of the Creative Commons Attribution license (<http://creativecommons.org/licenses/by/3.0/>).