Solving Difficult Instances of Boolean Satisfiability in the Presence of Symmetry

Fadi A. Aloul, *Student Member, IEEE*, Arathi Ramani, Igor L. Markov, and Karem A. Sakallah, *Fellow, IEEE*

***Abstract*—Research in algorithms for Boolean satisfiability (SAT) and their implementations (*Goldberg and Novikov*, 2002), (*Moskewicz et al.*, 2001), (*Silva and Sakallah*, 1999) has recently outpaced benchmarking efforts. Most of the classic DIMACS benchmarks (*ftp:dimacs.rutgers.edu/pub/challenge/sat/benchmarks/cnf*) can now be solved in seconds on commodity PCs. More recent benchmarks (*Velev and Bryant*, 2001) take longer to solve due to their large size, but are still solved in minutes. Yet, relatively small and difficult SAT instances must exist if $\mathrm{P} \neq \mathrm{NP}$. To this end, our paper articulates SAT instances that are unusually difficult for their size, including satisfiable instances derived from very large scale integration (VLSI) routing problems. With an efficient implementation to solve the graph automorphism problem (*McKay*, 1990), (*Soicher*, 1993), (*Spitznagel*, 1994), we show that in structured SAT instances, difficulty may be associated with large numbers of symmetries. We point out that a previously published symmetry extraction mechanism (*Crawford et al.*, 1996) based on a reduction to the graph automorphism problem often produces many spurious symmetries. Our paper contributes two new reductions to graph automorphism, which extract all correct symmetries found previously (*Crawford et al.*, 1996) as well as phase-shift symmetries not found earlier. The correctness of our reductions is rigorously proven, and they are evaluated empirically. We also formulate an improved construction of symmetry-breaking clauses in terms of permutation cycles and propose to use only generators of symmetries in this process. These ideas are implemented in a fully automated flow that first extracts symmetries from a given SAT instance, preprocesses it by adding symmetry-breaking clauses, and then calls a state-of-the-art backtrack SAT solver. Significant speed-ups are shown on many benchmarks versus direct application of the solver. In an attempt to further improve the practicality of our approach, we propose a scheme for fast "opportunistic" symmetry extraction and also show that considerations of symmetry may lead to more efficient reductions to SAT in the VLSI routing domain.**

***Index Terms*—Backtrack search, clause learning, conjunctive normal form (CNF), graph automorphism, logic simplification, satisfiability (SAT), symmetries.**

I. Introduction

BOOLEAN satisfiability (SAT) is a pivotal problem in computer science with numerous applications that range from microprocessor verification [60] to field programmable gate array (FPGA) layout [46]. A one million dollar prize is offered by the Clay Institute for Mathematical Sciences for a complete, polynomial-time SAT solver or a proof that such an algorithm does not exist (the P-versus-NP problem). Additionally, industrial applications motivate intensive research in SAT algorithms that quickly solve real-life instances. The fundamental framework for state-of-the-art SAT algorithms was laid out in the 1960s, but a number of recent improvements in algorithms and implementation techniques [45], [50] have led to performance breakthroughs. Most DIMACS challenge benchmarks [22] from the early 1990s are now solved in seconds on commodity PCs. Recently posted SAT benchmarks [60] take somewhat longer to solve (minutes), but that is primarily due to their enormous size (50 MB+ files, etc.). With the exception of artificially constructed families of benchmarks, it appears that SAT can be solved in polynomial time "for practical purposes." It is well known that the dominant backtrack solvers, such as GRASP [50], CHAFF [45], and BerkMin [11] do not perform well on randomly created 3-SAT instances with ≈ 4.3 clauses per variable [52]. However, such instances are not common in practical applications because they have little structure. The relative ease of structured instances from certain applications was explained [9], [47], and generic ways to exploit certain types of structure were proposed [2].

A. Difficult SAT Benchmarks

Our paper addresses both benchmarking and algorithmic aspects of SAT research. Given the excellent performance of existing SAT solvers, there is no room for improvement on easy benchmarks, and we focus instead on difficult instances. Since the work of Haken and Urquhart [58] on lower bounds for resolution and backtracking algorithms for SAT, several instance families have been known to require exponential time for Davis–Putnam [20] and Davis–Logemann–Loveland [21] (DP/DLL) solvers and their derivatives. For example, a recent lower bound for the pigeonhole problem is $\Omega(2^{n/20})$[7] where n is the number of holes. The pigeonhole problem can be quickly solved by induction, but the proof system behind backtrack solvers (resolution) is rather restrictive and does not allow polynomial-sized proofs for pigeonhole instances. Short proofs without induction exist if the use of symmetry is allowed [32], [59]. Another family of difficult instances was constructed by Tseitin and Urquhart in terms of expander graphs and, unlike the pigeonhole instances, can accommodate considerable randomness [57], [58]. Solving these instances takes a long time using modern SAT solvers such as CHAFF and BerkMin (see Tables IV and V), but their relevance to application domains (e.g., electronic design automation (EDA)

Manuscript received September 6, 2002; revised December 11, 2002. This work was supported in part by the DARPA/MARCO Gigascale Silicon Research Center, in part by an Agere Systems/SRC Research fellowship, and in part by a fellowship from the ACM/IEEE Design Automation Conference. This paper was previously presented at the ACM/IEEE Design Automation Conference, New Orleans, LA, in June 2002. This paper was recommended by Associate Editor J. H. Kukula.

The authors are with the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109-2122 USA (e-mail: faloul@eecs.umich.edu; ramania@eecs.umich.edu; imarkov@eecs.umich.edu; karem@eecs.umich.edu).

Digital Object Identifier 10.1109/TCAD.2003.816218

and software verification) is not clear. While lower bounds for SAT are often proven for unsatisfiable instances, it remains to be seen whether practical satisfiable instances can be difficult for the best solvers. To this end, the work in [1] contributed constructions of artificial randomly generated difficult satisfiable instances.

Our paper demonstrates EDA-related SAT instances, both satisfiable and unsatisfiable, that are very difficult for their size. Observe that an easy instance of any size can be made difficult by adding a small difficult instance to it and connecting the two by inconsequential clauses to defeat partitioning.

B. Relevance of Graph Automorphism to SAT

Over many years, empirical algorithms research in many domains identified a number of fundamental problem formulations, such as Boolean satisfiability, and mustered significant efforts to solve them efficiently. State of the art is gauged by optimized solver implementations ("engines"). Performance breakthroughs are often due to novel algorithmic ideas, leaner implementations, or the ability to apply a highly optimized engine in a novel way. In this paper, we observe that graph automorphism engines can be applied to the satisfiability problem in certain cases. Additionally, we think that there may be significant room for future improvement given that: 1) the graph automorphism problem is not thought to be NP-complete, thus, potentially easier than SAT and 2) much less new research was done in recent years on the analysis and design of high-performance engines for graph automorphism (such work includes [40] and [44]). To be precise, in this paper, we will be dealing with the colored variant of the graph automorphism problem that can be easily extended to hypergraphs.

Besides complexity-theoretic connections between variants of Boolean satisfiability, symmetries, and the hypergraph automorphism problem [4], [38], several pre-2000 publications suggested that "breaking symmetries" in conjunctive normal form (CNF) formulas can speed up SAT solvers [8], [13], [14], [18], [19], [40]. Symmetries of a CNF formula include clause-preserving permutations of variables. Such permutations may involve arbitrarily many variables at once, e.g., a complete cyclic shift. In this paper, we do not address permutations that change the CNF formula but leave unchanged the Boolean function it represents.[1] However, if such symmetries are found by other techniques [30], our proposed methods can process them in the same way as symmetries of the CNF formula. Similarly, many of the publications we cite do not deal with symmetry extraction, but rather assume that symmetries of the Boolean function are given. Using this assumption, two main directions were explored: 1) preprocessing the original CNF formula by adding symmetry-breaking clauses that do not affect satisfiability but speed up search [19] and 2) extending SAT solvers, particularly those based on backtracking, to dynamically use symmetries during the search process [6], [14], [35], [48]. In this paper, we pursue the preprocessing approach due to its simplicity, but will outline how our techniques can be applied within a backtracking solver for increased efficiency.

[1]Such permutations can be called "semantic" symmetries, in contrast with the narrower class of "syntactic" symmetries that leave the CNF formula unchanged.

C. Empirical Efficiency Challenges

Most prior work on symmetries in SAT predates recent breakthroughs in SAT solvers and typically uses several carefully constructed instances to illustrate their approaches or do not show convincing empirical results at all. For example, Crawford *et al*. suggest in [19] that symmetry-based techniques allow the pigeonhole instances to be solved in polynomial time, but their empirical data [19, Fig. 3] do not support this suggestion. In the course of more recent work [35], [54], specific families of CNF formulas with extremely high numbers of symmetries were successfully attacked. Yet, it remains unclear whether the performance of leading edge SAT solvers can be improved, via the use of symmetries, on large CNF families of practical significance. In principle, the overhead due to symmetry extraction and usage may outweigh the benefits, and it remains to be seen whether useful CNF formulas have many symmetries. Pólya (1937), Erdös, and Rényi (1963) proved that a random graph on n vertices has *no symmetries* with probability $1-\binom{n}{2}2^{-n-2}(1+o(1))$ [5, p. 1461]. This claim can be extended to CNF formulas, but structured real-world instances may have richer symmetries. Indeed, Boolean functions arising in the design of hardware systems often have many symmetries [10], [30], and the overall number of functions of n variables with nontrivial symmetries grows double, exponentially. On the other hand, for a function with exponentially many symmetries, trying to explicitly use all symmetries may defeat the purpose of speeding up search [19]. Despite these pitfalls, symmetry-based approaches have been useful in model checking [16], [24], [28], nonstandard SAT solvers [25], hardware verification [40], software verification [12], logic synthesis [10], [31] and DSP algorithms [23]. Some researchers limited the notion of symmetry to swaps of variables [23] or subsets of variables [31] to achieve efficiency. Other authors [10], [48] limited the notion of symmetry to negations of single variables or subsets of variables and referred to those restricted classes as *autosymmetries* or *phase-shift symmetries*.

D. Our Contributions

In this paper, we study and fully automate a flow that starts with a CNF formula in the DIMACS format and finds all of its symmetries within a very general class, including all permutational symmetries, variable negations, and their compositions. In this flow, all symmetries are first captured implicitly, in terms of irredundant group generators, which always guarantees exponential compression. The CNF formula is then preprocessed by adding symmetry-breaking clauses that do not affect satisfiability. A black-box SAT solver is subsequently applied to the preprocessed CNF instance to produce the final answer; any satisfying assignment to this instance is (or corresponds to) a satisfying assignment of the original instance, and if the preprocessed instance is unsatisfiable then so is the original instance. The flow is illustrated in Fig. 1.

We propose new techniques for symmetry extraction and empirically compare them with previously proposed constructions. We also propose a novel construction of symmetry-breaking clauses, which is much more economical than that in [19]. Also, it directly applies to the compressed representation of all symmetries in the format produced by graph-automorphism software [42], [43], [55], [56].

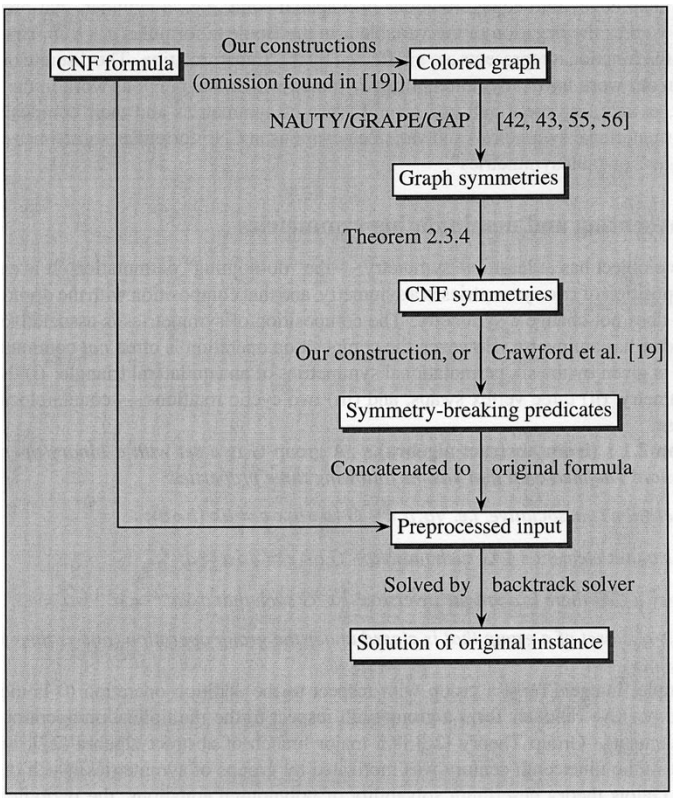


Fig. 1. Preprocessing-based flow for symmetry breaking studied in this paper. Our construction of symmetry-breaking predicates improves upon that from [19].

Our empirical results show significant overall performance improvements on CNF instances arising in EDA applications, as well as on highly randomized, provably difficult Urquhart benchmarks [58] that are related to Tseitin formulas [57] used to prove lower bounds on the size of resolution proofs. Two extensions are proposed to speed up symmetry extraction. One is opportunistic symmetry extraction, where only some symmetries are found. The other extension pursues domain-specific symmetries and leads to improvements of SAT formulations by adding domain-specific symmetry-breaking clauses. Thus, generic symmetry extraction is avoided by creating symmetryless SAT instances that can be solved quickly.

The remaining material is organized as follows. Symmetry extraction is described in Section II and symmetry-breaking in Section III. Section IV discusses constructions of SAT benchmarks. Our empirical results are presented in Section V and further extensions in Section VI. Section VII concludes our paper and discusses our future directions.

II. Symmetry Extraction

In general, a symmetry of a discrete object is a reversible transformation of its components that leaves the object unchanged. This can be taken as an informal definition, and more rigorous definitions will be given below for specific structures. Examples include permutations of graph vertices that map edges into edges, rotations of a spatial solid, e.g., a cylinder, that preserve its shape, as well as the negation of the variable a in the Boolean formula $(a + a')b$, since the formula and the function it represents are unaffected by this transformation. The discrete objects considered in our paper have only finitely many symmetries. Unlike previous work in the field, we consider, extract, represent, and use several types of symmetries and their compositions, including permutational symmetries and variable negations in CNF formulas, sometimes called "phase changes" or "autosymmetries."

A. Representing and Manipulating Symmetries

Every discrete object has at least one symmetry—the "do-nothing" permutation. It is easy to see that composition of two symmetries is a symmetry, and that composition with the do-nothing permutation does not change a symmetry. The composition of symmetries is associative, and every symmetry has an inverse. However, the composition operation is often *not* commutative. An example is given by the six permutational symmetries of an equilateral triangle: 1) the do-nothing symmetry; 2) three vertex swaps; and 3) two cyclic rotations—counterclockwise and clockwise.

Definition 2.1.1 (From Abstract Algebra): A group G *is a set with a binary operation ("multiplication") defined on it that has the following three properties:*

- *the operation is associative, i.e.,* $\forall a, b, c \in G\ (a \circ b) \circ c = a \circ (b \circ c)$;
- *there is a* unit *element* $e \in G$ *such that* $\forall a \in G\ a \circ e = e \circ a = a$;
- *for every* $a \in G$ *there is a unique* inverse $a^{-1} \in G$ *such that* $a \circ a^{-1} = a^{-1} \circ a = e$.

A subgroup *is a subset of a group that is closed under the group operation (and is, therefore, a group itself).*

For example, integers form a group with respect to the addition operation (0 is the unit element) and positive rationals form a group with respect to the multiplication operator (1/1 is the unit element). Group Theory [26] is a major branch of abstract algebra [27] and its development in the nineteenth century was motivated by groups of symmetries. Such diverse areas as the Galois theory describing solvability of polynomial equations, the periodic table of chemical elements, and Special Relativity involve analyses of groups of symmetries. In this paper, we will only deal with groups of symmetries whose elements can be thought of as permutations of finite sets. This obviously restricts us to finite groups. A permutation can be represented by cycles, e.g., (23)(567) represents a permutation on a set of at least seven marks (elements). This permutation swaps marks 2 and 3, cyclically permutes marks 5, 6, and 7 in that order, and leaves unchanged all other marks, e.g., 1 and 4.

Computational group theory (CGT), which started around 1911, is one of the oldest and most developed branches of computational algebra [53]. The flourishing of CGT began in the 1960s and great strides were made in the 1990s with the development of the GAP package ("Groups, Algebra and Programming") [56]. A major source of efficiency in CGT comes from the notion of *irredundant sets of generators* of a group.

Definition 2.1.2: A set of generators *consists of group elements such that any other group element can be composed of generators and their inverses. A generator is* redundant *if it can be expressed in terms of other generators. An* irredundant set of generators, *by definition, does not contain redundant generators.*

(Lagrange) Theorem 2.1.3 (from Elementary Group Theory) [26], [27]: The size of any subgroup H of any finite group G must divide the size of G.

Corollary 2.1.4: For any group G with $N > 1$ elements, any irredundant set of generators contains at most $\log_2 N$ *elements.*

Proof: Observe that any proper subgroup must be at least twice as small compared to the group. Given a set of n irredundant generators $x_1, \ldots, x_n$, consider a chain of subgroups G_k for $k = 1 \ldots n$, where G_k is generated by $x_1, \ldots, x_k$. By construction, G_k is a proper subgroup of G_{k+1}, and as such must be at least twice as small. Therefore, the size of $G_n = G$ must be at least 2^n. □

For example, the $k!$ permutations on k marks can be generated by (12) and $(12..k)$ or by $(12), (23), \ldots, (k-1\ k)$. Thus, representing groups by sets of irredundant generators *always ensures exponential compression*. CGT provides efficient algorithms (due to Sims, Knuth, Babai, and others) for manipulating groups represented by sets of generators, without decompression. Therefore, an intelligent algorithm for symmetry extraction may return a small set of generators rather than list all symmetries.

Definition 2.1.5: A mapping f: $G_1 \rightarrow G_2$ between two groups is a homomorphism *if and only if for any $a \in G_1$ and $b \in G_1$, we have $f(a \bullet b) = f(a) \circ f(b)$, where $\bullet$ and $\circ$ are group operations in G_1 and G_2, respectively. A homomorphism for which an inverse mapping exists that is also a homomorphism, is called an* isomorphism. *If an isomorphism exists between G_1 and G_2, the two groups are called* isomorphic. *An isomorphism of a group with itself is called* automorphism *of that group and can be thought of as a symmetry of the group.*

Automorphisms can be composed, and form a group under this operation.

It is easy to see that if f is a homomorphism, then $f(a^{-1}) = f(a)^{-1}$. An isomorphism cannot map two different group elements to one. Additionally, the notion of isomorphism defines an equivalence relation and is useful to compare groups formally defined over different sets. In simple terms, isomorphic groups have "the same structure." Therefore, when looking for a group of symmetries of some objects, it may be convenient to find an isomorphic group instead. Since groups are often described by sets of generators, it is important to know that isomorphisms preserve such descriptions.

Theorem 2.1.6: Any group isomorphism maps sets of generators to sets of generators, and maps irredundant sets of generators to irredundant sets of generators.

Proof: If any element $h \in G_1$ can be written as a product of elements of a generating set or their inverses $h = g_1 \bullet g_2 \bullet \cdots \bullet g_n$, then a homomorphism f: $G_1 \rightarrow G_2$ will preserve such expressions in G_2: $f(h) = f(g_1) \circ f(g_2) \circ \cdots \circ f(g_n)$. Since every isomorphism has an inverse, any element $k \in G_2$ can be mapped back to G_1, where its preimage can be decomposed into a product and then mapped back to G_2. This constructs a decomposition of k into a product of the images of elements of a generating set in G_1 and their inverses.

Now, consider a pair of sets of generators that are mapped to each other by an isomorphism, they must have the same cardinality. Assume that one of them has a redundant element that can be expressed in terms of remaining elements. Since such an expression is preserved by an isomorphism, the image of this element must be redundant in the other set of generators. □

B. Colored Automorphism Problems

Combinatorial objects are commonly represented by graphs. Therefore, we study symmetries of graphs first.

Definition 2.2.1: Given two graphs, an isomorphism *is a 1-to-1 mapping between the vertex sets of the two graphs that maps edges to edges. Given a graph, a* symmetry *(also called an* automorphism*) is a permutation of its vertices that maps edges to edges. In the case of directed graphs, edge orientations must be preserved.*

Definition 2.2.2: In the graph automorphism *problem, one seeks all symmetries of a given graph, e.g., in terms of group generators. The* decision version *of this problem tests for the presence of nontrivial automorphisms.*

It is known that all graphs, except for an exponentially small family, have *no symmetries* [5, p. 1461]. No general worst-case polynomial-time algorithms are known for this problem, but it is commonly believed not to be NP-complete [33]. Polynomial-time algorithms are available in many special cases [5, p. 1511], in particular for graphs of bounded degrees [37], [3]. Observe that graphs of bounded degree arise in many practical applications because the objects involved (logic gates in VLSI chips, facts stored in knowledge bases, etc.) are interconnected sparsely. In contrast, Boolean Satisfiability instances of bounded degree, e.g., 3-SAT, are known to be NP-complete and 3-SAT instances may be quite difficult in practice even if every literal participates in only several clauses [52]. Generic algorithms for the graph automorphism problem [42] are based on linear-time partition refinement passes, followed by backtrack search. A simple version of partition refinement completes in three passes and does not require follow-up backtracking for all but an exponentially small family of graphs [5, p. 1513]. However, exponential worst cases have been constructed even for very sophisticated versions [42], both theoretically and empirically [44].

The graph automorphism problem may be constrained by vertex labels—symmetries must map each vertex into a vertex with the same label. Label constraints are computationally easy and can be formally reduced to plain graph automorphism. Labels are often expressed by integers and called colors (no relation to *graph coloring*). Another extension is to colored *hypergraphs*—symmetries must map hyperedges to hyperedges (of the same cardinality because no two vertices can map to one). The colored hypergraph automorphism problem reduces to the colored graph automorphism via the bipartite graph of the hypergraph. This graph contains a vertex for each hypergraph vertex and hyperedge, and connects them with edges according to the hypergraph's incidence relation. Graph vertices in the hyperedge part are painted with a new color, and other vertices retain their original colors.

Brendan McKay implemented a practical algorithm for graph automorphism [42] in a software package called NAUTY [43], which has been continually improved for the last 20 years.[2] NAUTY has been integrated into the CGT system GAP [56] by means of the GRAPE package [55]. This integration enables efficient group-theoretic operations on the results returned by NAUTY and facilitates some of our proposed algorithms. In

[2]NAUTY version 2.0 was released in 2001.

1998, Manku *et al.* [40] claimed speed-ups over a pre-2.0 version of NAUTY in the context of hardware verification. However, their code is not generic (built into a larger system) and is no longer supported. Finally, we observe that the run-time of existing graph automorphism programs, e.g., NAUTY, typically increases with growing numbers of vertices and symmetry generators found, but may decrease with growing numbers of vertex colors and, sometimes, graph edges.

C. CNF Symmetries via Graph Automorphism

The problem of extracting symmetries of a CNF formula is reduced to the colored graph automorphism problem. The main idea behind such reductions is to find a colored graph whose symmetry group is isomorphic to the symmetry group of the CNF formula. Related constructions are described in [18] and [19] for permutational symmetries, and we draw upon them in our work. Consider a CNF formula with V variables and C clauses, of which, C_2 are binary and C_x have two or more literals (clauses with fewer than two literals can be removed by preprocessing). In quotations, the word "theories" refers to CNF formulas. From [18, p. 3]:

> Now consider reducing symmetry extraction to graph isomorphism. We show the mapping for propositional theories (…). First note that we can "type" the nodes in the graphs, and only allow isomorphisms which preserve type (…), without increasing the difficulty of the isomorphism problem. We use five types of nodes: nodes for positive literals, nodes for negative literals, *inverse* nodes, nodes for clauses and *goal* nodes. We first link (the node for) each literal p to an inverse node and then link this inverse node to (the node for) $\neg p$. These links ensure that any graph isomorphism preserves negation. We then create a node for each clause and link it to the literals appearing in the clause. These links force graph isomorphisms to map clauses to clauses. Finally, recall that we are required to find a θ which maps p to q. To force this we create two copies of the graph for the theory. In the first we give p the type *goal* and in the second we give q the type *goal*. This typing forces any isomorphism between the two graphs to map p to q. One can then show that an isomorphism between the graphs exists if and only if the theory contains a simple symmetry mapping p to q.

The author then concludes that the *decision version* of the CNF symmetry detection problem is polynomial-time solvable if the length of the longest clause and the number of occurrences of the most common literal are bounded by a constant. That is because the degree of graph vertices is bounded by that constant, in which case, the graph automorphism problem is poly-time solvable [5], [37]. If applied literally, the proposed construction only addresses symmetries that map p to q for particular p and q, rather than arbitrary symmetries. In order to find even a single nontrivial symmetry, one may need to traverse all pairs. Thus, no isomorphism of symmetry groups is claimed in [18], and no empirical results are reported. Additionally, we observe that for a formula with V variables and C clauses, this construction produces a graph with $6V + C$ vertices. Given that run-time of graph automorphism programs, e.g., NAUTY, grows superlinearly in terms of the number of vertices, more economical constructions (see below) can significantly reduce run-time.

Despite being impractical, the construction from [18] was apparently the first to introduce fundamental elements, now used by more competitive constructions, including ours. We emphasize as particularly important:

- the modeling of variables by pairs of positive-literal and negative-literal vertices;
- the modeling of each clause by a vertex connected to respective literal vertices by edges;
- connecting positive- and negative-literal vertices to enforce Boolean consistency.

Additional useful elements were introduced in [19, p. 7]:

> The input theory is converted into a graph such that the automorphisms of the graph are exactly the symmetries of the theory. This is done using the construction in [Crawford, 1992]. There are three "colors" of vertices in this graph: vertices representing positive literals, those representing negative literals, and those representing clauses. Graph automorphisms are constrained to always map nodes to other nodes of the same color. We also add edges from each literal to each clause that it appears in. These edges (together with the node colorings) guarantee that automorphisms of the graph are the symmetries of the theory. *Footnote 5*: For efficiency we special-case binary clauses by representing $x \wedge y$ with a link directly from x to y (instead of creating a node for the binary clause and linking x and y to it). This is important because some of the instances we consider have a huge number of binary clauses and some of the algorithms that follow are quadratic, or worse, in the number of nodes.

The reference [Crawford, 1992] in this quotation is the same as reference [18] in our paper, but the construction appears different from that cited above.[3] In fact, this formulation seems to omit the enforcement of Boolean consistency. This leads to the generation of many spurious symmetries. For example, the formula $(a + b)$ has two symmetries: 1) the do-nothing symmetry and 2) the transposition (ab). The graph built by the above procedure has two positive-literal vertices, two negative-literal vertices and one clausal vertex connected to the positive-literal vertices by two edges. Since no negative literals are used, the respective vertices are disconnected and can be mapped to each other even if positive-literal vertices are fixed. There are four symmetries. One of them is the swap (transposition) of $\bar{a}$ and $\bar{b}$ with a and b fixed. It violates Boolean consistency. Notably, in [19], this construction is described in Section VII on empirical results, next to a discussion of pigeonhole and n-queens benchmarks. However, it produces spurious symmetries even when applied to pigeonhole benchmarks, starting with hole-2.

On the positive side, this construction produces a graph with $2V + C_{\times}$ vertices—a marked improvement over [18]. We also found very useful in practice the idea to model each binary clause by one edge rather than by one vertex and two edges. The proposed construction can be corrected by adding, for each

[3]Both papers [18] and [19] are downloadable [Online] from http://citeseer.nj.nec.com/cs and also from http://www.cirl.uoregon.edu/crawford/papers/papers.html .

variable, a vertex of color 4 and connecting it to the positive- and negative-literal vertices for the same variable (these nodes were called *inverse* nodes in [18]). We implemented this corrected version, and report empirical results for it. Similar to [18], the reduction from [19] and its corrected version cannot find phase-shift symmetries because it colors positive and negative literals with different colors.

In this paper, we propose several reductions of CNF symmetry extraction to graph automorphism, all of which allow extracting phase-shift symmetries and their compositions with permutational symmetries. One of our constructions produces $2V + C_{\times}$ vertices and never finds spurious symmetries, but requires double edges that are not supported by the graph automorphism software NAUTY [43] used in our experiments. Another proposed construction produces $2V + C_{\times} + \min\{C_2, V\}$ vertices and never finds spurious symmetries. The third construction produces $2V + C_{\times}$ vertices, is implementable with NAUTY, produces no spurious symmetries on our benchmarks, and allows a trivial check for spurious symmetries in general. Since this construction is often the fastest in practice, we characterize CNF formulas on which it produces spurious symmetries and show how spurious symmetries can be removed.

We first preprocess a given CNF formula to remove any clauses with fewer than two literals. If there is an empty clause, the formula is immediately declared unsatisfiable and the search for the symmetries of the formula becomes pointless. If there are one-literal clauses, they can be eliminated in linear time by repeatedly 1) recording implied truth assignments [clause (a) implies $a = 1$, clause $(\bar{a})$ implies $a = 0$]; 2) eliminating the one-literal clauses; 3) substituting the implied values of relevant variables, thus eliminating the variables; and 4) simplifying each affected clause independently. This process will either prove the original formula satisfiable/unsatisfiable, or result in a smaller formula where every clause has at least two literals.

Given a CNF formula where every clause contains at least two literals, we represent every variable by two vertices that correspond to its positive and negative literals. We represent every nonbinary clause by a single vertex, and connect that vertex to the vertices representing literals in that clause. Binary clauses are represented by double edges connecting their respective literals. Clausal vertices are painted color 1 and literal vertices with color 2. Since vertices representing positive and negative literals in our graph are of the same color, we need to ensure Boolean consistency and mate vertices of opposite literals by single edges. Observe that no symmetry can map a single edge to a double edge, thus, there is no risk of mapping a Boolean consistency edge to a binary-clause edge. This construction results in a graph with $2V + C_{\times}$ vertices. It corrects the reduction from [19] without increasing vertex counts and has the added advantage of extracting phase-shift symmetries (subsets of negated variables, e.g., $a \mapsto \bar{a}$) and their compositions with permutational symmetries. We refer to this construction as $2 \times$ EDGES.

Unfortunately, the graph automorphism program NAUTY [43] used in our experiments cannot represent double edges. Therefore, we must seek another mechanism to distinguish Boolean consistency edges from binary-clause edges. A straightforward solution is to split every Boolean consistency edge into two edges by an added vertex of color 3 (one per edge). Alternatively, we can split binary-clause edges, which in some cases may be a better option. In fact, we can split the less numerous of the two types of edges, which yields $2V + C_{\times} + \min\{C_2, V\}$ vertices. Because three colors are used, this construction is referred to as MIN3C.

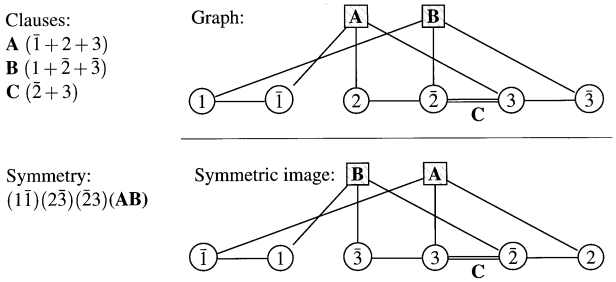


Fig. 2. CNF formula with three clauses—A, B, and C,—and three variables is converted into a bicolored graph for symmetry extraction purposes. The two-literal clause C is represented by one edge (double-line) while larger clauses A and B are represented, each, by a vertex and three edges. Any symmetry must map C $\mapsto$ C, and therefore, this instance has only one nontrivial symmetry $(1\bar{1})(2\bar{3})(\bar{2}3)(\mathrm{AB})$.

A far less obvious solution is *not to make an explicit distinction* between the two types of edges, but represent both Boolean consistency and binary clauses by single edges. Since we first described this construction at the 2002 Design Automation Conference, we refer to it as DAC'02. Fig. 2 shows an example. In general, there are $2V + C_{\times}$ vertices, but the analysis of this construction is far more complex than that of the constructions described above. However, our efforts are justified by the often-superior empirical performance of this construction. Before we proceed with formal results, let us articulate the correspondence between 1) the variables and clauses in a given CNF formula and 2) the vertices and edges of the bicolored graph we build. Every variable corresponds to exactly two vertices of color 2. Every vertex of color 2 corresponds to a variable, and every vertex of color 1 corresponds to a clause. Every clause with more than two literals corresponds to a vertex of color 1, and every two-literal clause corresponds to an edge between two vertices of color 2. There are no edges connecting vertices of color 1, but every vertex of color 2 is connected to that of its complement literal by an edge, and there can be edges connecting pairs of vertices of different colors.

Definition 2.3.1: A circular chain of implications *over the variables $x_1, x_2, \ldots, x_N$ is a set of N binary clauses equivalent to $(y_1 \Rightarrow y_2)(y_2 \Rightarrow y_3)\ldots(y_{N-1} \Rightarrow y_N)(y_N \Rightarrow y_1)$, where for each k from $1..N$, $y_k = x_k$ or $y_k = \bar{x}_k$.*

Observe that the clause $(\bar{y}_k + y_{k+1})$ is equivalent to $(y_k \Rightarrow y_{k+1})$ and also to $(\bar{y}_{k+1} \Rightarrow \bar{y}_k)$. In terms of specific values, we have $(y_k = 1) \Rightarrow (y_{k+1} = 1)$ and $(y_{k+1} = 0) \Rightarrow (y_k = 0)$. For each k, one of the two possible values of y_k triggers an implication sequence, and, thus, unambiguously determines the values of all literals involved. In the remaining case, none of the variables assume the value that triggers an implication in the circular chain. Therefore, a circular chain of implications allows only two satisfying solutions.

Theorem 2.3.2: Assume that a given CNF formula does not contain a circular chain of implications over any subset of its

variables. Then, with respect to the proposed construction of the colored graph from a CNF formula, the symmetries of the formula correspond one-to-one to the symmetries of the graph.

The practicality of the assumption is discussed after Corollary 2.3.4 as follows.

Proof: It is not hard to see that every permutational symmetry of the initial formula (i.e., a permutation of variables that maps clauses to clauses) corresponds to a colored symmetry of the bicolored graph we built. Such a graph symmetry will map vertices to vertices of the same color and edges to edges. In particular, if a maps to b, then $\bar{a}$ maps to $\bar{b}$ and the edge $a\bar{a}$ maps to the edge $b\bar{b}$. Edges between vertices of color 2 will always map to edges between vertices of color 2, and the same can be said about edges between vertices of different colors. Phase-shift symmetries of the original formula also correspond to colored graph symmetries. For example, $a \mapsto \bar{a}$ will induce a swap between the vertices a and $\bar{a}$, leaving the edge $a\bar{a}$ in place and swapping any existing edges ac and $\bar{a}c$ for a clausal vertex c. An immediate consequence is that every composition of permutational and phase-shift symmetries of the original formula correspond to a colored graph symmetry. For example, if a is symmetric to $\bar{b}$, then $a \mapsto \bar{b}$ and $\bar{a} \mapsto b$ so that the edge $a\bar{b}$ maps to $\bar{a}b$.

Our next observation is that given a colored graph symmetry that corresponds to some CNF symmetry, we can always uniquely reconstruct the CNF symmetry as long as the correspondence between variables and vertices of color 2 is available. This is also shown by first considering purely permutational symmetries, then phase-shift symmetries, and then their compositions. A graph symmetry that corresponds to a permutational CNF symmetry must map positive-literal vertices to other such. Therefore, we can restrict the graph symmetry to this subset of vertices, thus producing a permutation of CNF variables. A graph symmetry that corresponds to a phase-shift CNF symmetry must either preserve a given literal vertex or map it to the complement-literal vertex, preserving the edge between them. Therefore, a list of positive-literal vertices that are not preserved uniquely identifies a phase-shift CNF symmetry. To reconstruct a CNF symmetry that is a composition of permutations and phase-shifts, we distinguish 1) positive-literal vertices that map to positive-literal vertices from 2) positive-literal vertices that map to negative-literal vertices. In each case, a given CNF variable is mapped to another variable, possibly with a follow-up negation. By ignoring the follow-up negations, we reconstruct the purely permutational component of the CNF symmetry. The phase-shift component, i.e., variables to be negated before the permutation is applied, can be reconstructed by listing positive-literal vertices that map to negative-literal vertices.[4]

Perhaps, the least trivial property of the proposed reduction to graph automorphism is that every colored symmetry of the graph corresponds to a symmetry of the original formula. To prove this, we show that the reconstruction procedure from the previous paragraph can be successfully applied to any colored graph symmetry. A vertex permutation is a colored symmetry if and only if 1) vertices are mapped to vertices of the same color and 2) edges are mapped to edges. This is consistent with CNF symmetries' mapping variables to variables and clauses with more than two literals to such clauses. However, it is more difficult to prove Boolean consistency, i.e., $\forall\, a, b\ (a \mapsto b) \Rightarrow (\bar{a} \mapsto \bar{b})$, where a and b are *literals*. This is easy in the absence of 2-literal clauses because all edges connecting vertices of color 2 are Boolean consistency edges of the form $\bar{a}a$. Since every such edge can only map to another such edge, $(a \mapsto b)$ leave no choice for $a\bar{a}$ but to map to $b\bar{b}$ because $b\bar{b}$ is the only edge that connects b to another vertex of color 2. This simple proof also applies if the two-literal clauses are represented by vertices, rather than by edges as in Fig. 2.

The difficulty in the general case is due to our modeling of two-literal clauses by edges that connect vertices of color 2. Such edges may potentially map to Boolean consistency edges, and our task is to prove such a mapping impossible.

We first present the following lemma.

Lemma 2.3.3: Let $M = (V, R)$ be a perfect matching on a finite vertex set V and let its edges R be colored red. Let $F = (V, G)$ be some graph on V. Let its edges G be colored green, where $R \cap G = \emptyset$. Let $\Gamma = (V, R, G)$ be the graph on V formed by taking the disjoint union of edge sets R and G. In other words, $\Gamma = (V, E)$, where $E = R \cup G$.

If Γ has no cycles with edges of alternating colors, then *every* automorphism of Γ *must* preserve the color of every edge.

Proof: Suppose σ is an automorphism of under which, w.l.o.g., a red edge r_0 maps to a green edge g_0. Since the red edges form a perfect matching, the green edge g_0 must share each of its end points with exactly one red edge. Let these be r_1 and ${r_1}'$, respectively. Since r_0 maps to g_0, there must be two distinct edges, each of which shares *exactly* one end point with r_0, to map into r_1 and ${r_1}'$ under σ. Furthermore, these two edges must be green edges, since red edges cannot share end points with other red edges. Call these edges g_1 and ${g_1}'$. We now have two paths of alternating colors, P_0 and P_1. An edge in P_1 is the σ-image (of opposite color) of the corresponding edge in P_0

$$P_0 = ({g_1}', r_0, g_1) \ \text{ and } \ P_1 = ({r_1}', g_0, r_1).$$

${g_1}'$ and g_1 must share their other end points (that are not shared with r_0) with two red edges, say r_2 and ${r_2}'$. In turn, images of r_2 and ${r_2}'$ must be green edges g_2 and ${g_2}'$ extending from the terminals of the path P_1. In effect, we have extended paths P_0 and P_1 to

$$P_0 = ({r_2}', {g_1}', r_0, g_1, r_2) \ \text{ and } \ P_1 = ({g_2}', {r_1}', g_0, r_1, g_2).$$

Repeating the foregoing argument for g_2 and ${g_2}'$ and continuing in this manner, we can "grow" paths of alternating colors

$$P_0 = (\ldots, {g_3}', {r_2}', {g_1}', r_0, g_1, r_2, g_3, \ldots)$$

and

$$P_1 = (\ldots, {r_3}', {g_2}', {r_1}', g_0, r_1, g_2, r_3, \ldots).$$

By the finiteness of Γ, one of the paths must eventually close on itself (when the two edges extending the current path turn out

[4]If one should perform negations *after* the permutation is applied, then the negative-literal vertices that map to positive-literal vertices should be listed.

to be the same). This will give us a cycle of alternating colors, thus contradicting the hypothesis. □

The proof of Theorem 2.3.2 is concluded as follows. Consider the graph representation G of a CNF formula that includes *only* the vertices representing literals, Boolean consistency edges, and edges representing binary clauses. (We do not consider vertices representing nonbinary clauses and edges connecting literal vertices to them since we have already shown that they do not produce spurious symmetries). In this graph, Boolean consistency edges correspond to the red edges in Lemma 2.2.3, and binary clause edges correspond to the green edges. It is clear that the Boolean consistency edges form a perfect matching, since they cover all vertices, and each vertex is covered by exactly one edge.

Finally, we observe that a cycle of alternating edges in the above graph G corresponds to a circular chain of implications in the CNF formula. A cycle of alternating edges is equivalent to the following clauses:

$$(x_1+x_2)(x_2+\bar{x_2})(\bar{x_2}+x_3)(x_3+\bar{x_3})\ldots(\bar{x_n}+\bar{x_1})(\bar{x_1}+x_1).$$

Since any clause of the form $(\bar{a}+a)$ is true, we can eliminate all such clauses, and the resulting formula is the following circular chain of implications:

$$(\bar{x_1}\Rightarrow x_2)(x_2\Rightarrow x_3)(x_3\Rightarrow x_4)\ldots(x_n\Rightarrow \bar{x_1}).$$

We have, thus, proved that a spurious symmetry (mapping a clausal edge to a Boolean consistency edge) is possible *if and only if* a circular chain of implications exists. This is a contradiction. □

The $2\times$ EDGES reduction avoids spurious symmetries altogether by connecting positive-literal vertices to negative-literal vertices with double edges.

Theorem 2.3.4: Under the assumption of Theorem 2.3.2, the symmetry groups of the CNF formula and the bicolored graph are isomorphic.

Since a one-to-one homomorphism must be an isomorphism, one only needs to verify that the one-to-one mapping constructed in the proof of Theorem 2.3.2 is a homomorphism.

Corollary 2.3.5: Under the assumption of Theorem 2.3.2, sets of symmetry generators of the bicolored graph correspond one-to-one to sets of symmetry generators of the CNF formula.

In terms of practicality, we observe that failure of the assumption in Theorems 2.3.2 and 2.3.4 implies that in every satisfying assignment, the variables involved in the circular chain of implications can assume one of two different sets of values (models). We illustrate this by using the CNF formula $(a+\bar{b})(b+\bar{c})(c+\bar{a})$, which allows only two models (000 and 111) but has six symmetries (do-nothing, two three-cycles, and three variable swaps combined with negation of all variables). Yet, the graph produced by our construction is a hexagon having 12 symmetries (the so-called *dihedral group* D_6[26], [27]). Half of those are spurious as explained in Fig. 3.

From the practical standpoint, we note the following.

- Circular chains of implications do not arise in standard SAT models from many application domains. For example, they do not appear in equivalence checking of combinational circuits because combinational circuits are directed acyclic graphs.
- The presence of circular chains of implications does not invalidate our construction. As can be seen from the proof of Theorem 2.3.2, the only potential problem is spurious graph symmetries that do not correspond to any CNF symmetries. Since any application of Theorem 2.3.2 must convert symmetry generators returned by a graph automorphism problem into CNF symmetries, any spurious symmetry generators can be identified with minimal computational effort and minimal programming overhead.
- If some, but not all, symmetry generators are spurious, the nonspurious generators are still useful for symmetry breaking, while spurious generators can be discarded (this approach may not be ideal because spurious symmetries can generate nonspurious ones).
- Since the product of nonspurious symmetries cannot be spurious, there can be no spurious symmetries at all if none of the symmetry generators are spurious. In other words, if spurious symmetries exist, at least one generator must be spurious.
- Once a spurious symmetry generator is found, a circular chain of implications can be identified in linear time along the lines of analysis in the proof of Theorem 2.3.2. Since every circular chain of implications implies two sets of values for variables involved, circular chains of implications can be *removed* by introducing one Boolean variable to represent the two sets of values (old variables get eliminated).
- In applications where many spurious symmetries are expected and can slow down symmetry extraction, circular chains of implications can be identified in linear time *before symmetry extraction*, using depth-first search on a directed graph of binary clauses.

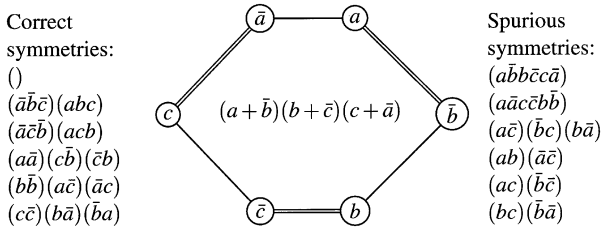


Fig. 3. Illustration of spurious symmetries: A CNF formula and its graph. Boolean consistency edges are shown by double lines, but are indistinguishable from other edges by graph automorphism software NAUTY which cannot handle double-edges. Therefore, the graph has 12 symmetries: 6 rotations and 6 axial flips. Only 6 of them—3 rotations and 3 flips—preserve Boolean consistency edges and correspond to symmetries of the CNF formula. The remaining 6 symmetries are spurious (the first three spurious symmetries shown are rotations and the remaining three are axial flips).

While the correctness of representing binary clauses with edges (Theorem 2.3.2) appears much harder to prove compared to the correctness of graph reductions proposed earlier, our construction reduces the number of vertices in the graph by the number of binary clauses in the CNF instance. Application-derived CNF instances typically have a significant proportion of binary clauses, and our construction DAC'02 leads to nontrivial run-time savings in practice. Table I summarizes the main properties of various reductions of CNF symmetry extraction

TABLE I
COMPARING REDUCTIONS OF CNF SYMMETRY EXTRACTION TO GRAPH AUTOMORPHISM. V IS THE NUMBER OF VARIABLES IN THE ORIGINAL CNF INSTANCE, C IS THE NUMBER OF CLAUSES, C_2 IS THE NUMBER OF BINARY CLAUSES, $C_\times = C - C_2$. THE $2\times$ EDGES REDUCTION IS NOT PRACTICAL WITH NAUTY BECAUSE NAUTY DOES NOT SUPPORT DOUBLE EDGES IN GRAPHS. CNF INSTANCES FOR WHICH THE DAC'02 REDUCTION FINDS SPURIOUS SYMMETRIES ARE CHARACTERIZED IN THEOREM 2.3.2

Reduction type	#Colors	#Vertices	Detects phase-shifts?	Finds spurious symmetries ?	Practical with NAUTY[43]?
[18]	5	$6V+C$	No	No	No
[19]	3	$2V+C_\times$	No	Many+often	No
2xEDGES	2	$2V+C_\times$	Yes	No	No
MIN3C	3	$2V+C_\times$ $+\min\{C_2,V\}$	Yes	No	Yes
DAC02	2	$2V+C_\times$	Yes	In rare cases + trivial check $\exists$	Yes

to graph automorphism. Additionally, we empirically compare MIN3C, DAC'02, the reduction from [19], and a corrected version of that reduction. In the corrected version, to ensure Boolean consistency, we add one extra node of color 4 for each variable and two edges connecting that node to the positive and negative literals of that variable.

Our testbed includes five sets of difficult benchmarks with nontrivial symmetries:

1) the hole-n benchmark set, available within the DIMACS collection [22];
2) randomized benchmarks Urq proposed by Urquhart [58], based on parity checks and expander graphs;
3) randomized benchmarks grout derived in this paper in the context of global grid-based routing for VLSI;
4) benchmarks FPGA derived in this paper in the context of detailed routing for field-programmable gate arrays;
5) recent benchmarks from the microprocessor verification domain [60].

Descriptions of all benchmark sets except for the Urq and microprocessor verification sets are given in Section IV. Our implementation of symmetry extraction uses the program NAUTY [43] version 2.0, shipped with the GAP package [56] version 4, release 3. Table II compares sizes of graphs produced by four constructions. We make the following observations.

- Because all of our benchmarks contain more binary clauses than variables, MIN3C generates exactly as many vertices and edges as the corrected version of the reduction from [19]. However, MIN3C produces one color less and extracts phase-shift symmetries.
- Graphs produced by MIN3C always have more vertices than those produced by DAC'02.
- DAC'02 and [19] produce graphs with the same numbers of vertices, but DAC'02 generates more edges because it ensures Boolean consistency.

Table III compares symmetry extraction run-time and the rounded number of symmetries (sizes of symmetry groups) discovered with each reduction. All run-times are recorded on a Linux workstation with a 1.2-GHz AMD Athlon and 1 GB of DDR RAM.

Several entries of the table with sizes of symmetry groups can be verified independently. For example, the number of symmetries in hole-n benchmarks is $n!(n+1)!$ because the symmetry group is the Cartesian product of S_n (holes can be permuted arbitrarily) and S_{n+1} (pigeons can be permuted arbitrarily). For $n=7$, this yields 203 212 800, which rounds off to $2.03e8$. Furthermore, we make the following observations.

- Except for the second (Urq) and the last (microprocessor verification) benchmark sets, the reduction from [19] produces more symmetries than other reductions. This is because it does not enforce Boolean consistency, and finds spurious symmetries. Urq benchmarks do not have permutational symmetries, as checked by the corrected version of [19]. The reduction from [19] cannot extract phase-shift symmetries.
- Except for the second and the last benchmark sets, the reductions MIN3C, DAC'02, and corrected [19] find the same numbers of symmetries. In particular, those three reductions produce correct numbers of symmetries for hole-n instances. This is consistent with the reduction from [19] being erroneous, as it discovers many spurious symmetries. In fact, the uncorrected [19] does not finish within the specified time limit on the FPGA instances, probably because it detects large numbers of spurious symmetries.
- Except for the second (Urq) benchmark set, the run-times of MIN3C and corrected [19] are comparable. This is expected because they generate equal numbers of vertices and edges, differing only in the number of colors. The run-times for Urq benchmarks are different because MIN3C leads to the discovery of more symmetries.
- DAC'02 is generally the fastest reduction. No other reduction generates fewer vertices, and DAC'02 does not discover any spurious symmetries on given benchmarks as its results always agree with MIN3C.

We explicitly verified that the symmetries discovered by MIN3C and DAC'02, but not by the two versions of [19], are phase-shift symmetries and their compositions with permutational symmetries. An implementation of the DAC'02 reduction is available in our software package Shatter that targets symmetry extraction and symmetry breaking for SAT. This package can be downloaded from http://gigascale.org/bookshelf/Slots/shatter/ .

TABLE II
COMPARISON OF REDUCTIONS IN TERMS OF SIZES OF GRAPHS PRODUCED

Instance	vari-ables	clau-ses	Previous work [19]		[19] corrected		Our work MIN3C		Our work DAC02	
			#vert	#edges	#vert	#edges	#vert	#edges	#vert	#edges
hole07	56	204	120	252	164	364	164	364	120	308
hole08	72	297	153	360	225	504	225	504	153	431
hole09	90	415	190	465	280	675	280	675	190	585
hole10	110	561	231	660	341	880	341	880	231	770
hole11	132	738	276	858	408	1122	408	1122	276	990
hole12	156	949	325	1092	481	1404	481	1404	325	1248
Urq3_5	46	470	560	2910	606	3002	606	3002	560	2956
Urq4_5	74	674	840	4270	914	4418	914	4418	840	4434
Urq5_5	121	1210	1450	7546	1571	7788	1571	7788	1450	7667
Urq6_5	180	1756	2122	10772	2292	11132	2292	11132	2112	10952
Urq7_5	240	2194	2672	13194	2912	13674	2912	13674	2672	13434
grout3.3-01	864	7592	2306	9024	3170	10752	3170	10752	2306	9888
grout3.3-03	960	9156	2558	10740	3518	12660	3518	12660	2558	11700
grout3.3-04	912	8356	2432	9864	3344	11688	3344	11688	2432	10776
grout3.3-08	912	8356	2432	9864	3344	11688	3344	11688	2432	10776
grout3.3-10	1056	10.8k	2796	12564	3852	14676	3852	14676	2796	13620
fpga23.21	725	6610	1954	12154	2679	13604	2679	13604	1954	12879
fpga23.22	759	7172	2046	13222	2805	14740	2805	14740	2046	13981
fpga24.22	792	7546	2134	13860	2926	15444	2926	15444	2134	14652
fpga24.23	828	8165	2231	15042	3059	16698	3059	16698	2231	15870
2pipe_1.ooo	834	7026	3851	14925	4685	16593	4685	16593	3851	15759
2pipe_2.ooo	925	8212	4133	17231	5058	19081	5058	19081	4133	18156
2pipe	861	6695	2621	12841	3482	14563	3482	14563	2621	13702
3pipe	2392	27533	7428	53620	9820	58404	9820	58404	7428	56012

III. SYMMETRY BREAKING

Symmetries induce equivalence classes on the set of truth assignments (in group theory, they are called *orbits*). Specifically, given a satisfying (unsatisfying) truth assignment, all other truth assignments to which it can be mapped by symmetries, must also be satisfying (unsatisfying). Therefore, for a complete SAT solver it suffices to reason about one representative from each such class. This restriction can be implemented by selecting unique representatives from every equivalence class and adding clauses that are only satisfied by those representatives. An earlier construction of such symmetry-breaking clauses [19] is based on a given ordering of variables. Its main ideas are 1) to order all elements from the solution space lexicographically and 2) to select the lexicographically smallest element from each equivalence class as its representative.

A. Previous Work

The lex-leader symmetry-breaking predicates described by Crawford *et al.* in [19] are built for a given group of permutational symmetries. Such predicates are conjunctions of smaller predicates for individual symmetries. Below, let n be the number of variables and $\mathrm{LL}(G)$ be the lex-leader symmetry-breaking predicate for the group G. Boolean variables x_k are traversed according to the original ordering

$$\mathrm{LL}(G) = \bigwedge_{\pi \in G} \mathrm{LL}(\pi) \tag{1}$$

$$\mathrm{LL}(\pi) = \bigwedge_{1 \le i \le n} C(\pi, i) \tag{2}$$

$$C(\pi, i) = \left[\bigwedge_{1 \le j < i} (x_j = x_j^{\pi}) \right] \Rightarrow (x_i \le x_i^{\pi}). \tag{3}$$

Example: Consider the formula $(a+\bar{c})(b+\bar{c})(a+b+c)(\bar{a}+\bar{b})$, from [19]. This formula has two symmetries, (ab) and the do-nothing symmetry. We compute $\mathrm{LL}(\pi)$ for $\pi = (ab)$ according to the equations above.

For $i = 1$ in (3), the null predicate $[\bigwedge_{1 \le j < i}(x_j = x_j^{\pi})\ ]$ is true. Also, since $\pi(a) = b$, we have $C(\pi, 1) = (a \le b)$.

For $i = 2$, $C(\pi, 2) = [(a = b) \Rightarrow (b \le a)]$.

For $i = 3$, $C(\pi, 3) = [(a = b \wedge b = a) \Rightarrow (c \le c)]$.

$C(\pi, 2)$ and $C(\pi, 3)$ are tautologies, therefore, we have $\mathrm{LL}(G) = (a \le b)$ as the symmetry-breaking predicate for this formula. Computing these predicates for the do-nothing formula also results in a set of tautologies which can be removed by simplification. However, we note here that *no general simplification procedure* is discussed in [19], so any lex-leader predicates derived from the equations above would have to be explicitly pruned to resolve tautologies. The constructions we propose require no simplification and use fewer clauses than the construction from [19] without simplification.

Theorem 3.1.1 [19]: For a group G acting on truth assignments, the truth assignments that satisfy $\mathrm{LL}(G)$ are the lexicographically smallest representatives from each class of truth assignments that can be mapped to each other by symmetries from G.

TABLE III
COMPARISON OF REDUCTIONS IN TERMS OF SYMMETRY EXTRACTION RUNTIME AND ROUNDED NUMBERS OF DISCOVERED SYMMETRIES. RUNTIMES ARE IN SECONDS ON A 1.2-GHZ AMD ATHLON WITH LINUX

Instance	variables	clauses	Previous work				Our work			
			[19]		[19] corrected		MIN3C		DAC02	
			time	#symm	time	#symm	time	#symm	time	#symm
hole07	56	204	0.25	1.47e70	0.0	2.03e8	0 .0	2.03e8	0.0	2.03e8
hole08	72	297	0.35	1.24e77	0.47	1.46e10	0.1	1.46e10	0.0	1.46e10
hole09	90	415	0.73	5.69e77	0.23	1.32e12	0.0	1.32e12	0.05	1.32e12
hole10	110	561	2.0	4.06e77	0.12	1.45e14	0.15	1.45e14	0.08	1.45e14
hole11	132	738	3.38	1.53e78	0.32	1.91e16	0.19	1.91e16	0.12	1.91e16
hole12	156	949	6.66	1.61e78	0.24	2.98e18	0.39	2.98e18	0.13	2.98e18
Urq3_5	46	470	0.05	1	0.21	1	0.51	5.37e8	0.39	5.37e8
Urq4_5	74	674	0.0	1	0.15	1	1.56	8.8e12	1.6	8.8e12
Urq5_5	121	1210	0.12	1	0.15	1	14.16	4.72e21	13.73	4.72e21
Urq6_5	180	1756	0.53	1	1.2	1	70.29	6.49e32	63.37	6.49e32
Urq7_5	240	2194	1.06	1	1.62	1	189.0	1.12e43	175.99	1.12e43
grout3.3-01	864	7592	109.1	8.28e77	16.36	8.71e9	15.44	8.71e9	4.86	8.71e9
grout3.3-03	960	9156	173.1	4.67e77	32.2	6.97e10	28.02	6.97e10	9.07	6.97e10
grout3.3-04	912	8356	143.5	1.10e78	21.85	2.61e10	19.65	2.61e10	7.01	2.61e10
grout3.3-08	912	8356	143.6	1.80e78	26.04	3.48e10	22.23	3.48e10	7.09	3.48e10
grout3.3-10	1056	10.8k	263.6	1.03e78	42.54	3.48e10	33.03	3.48e10	10.73	3.48e10
fpga23.21	725	6610	>1000	—	105	1.42e50	106	1.42e50	40.9	1.42e50
fpga23.22	759	7172	>1000	—	123	3.40e52	126	3.40e52	49.2	3.40e52
fpga24.22	792	7546	>1000	—	170	8.20e53	173	8.20e53	58.3	8.20e53
fpga24.23	828	8165	>1000	—	170	1.13e56	171	1.13e56	67	1.13e56
2pipe_1.ooo	834	7026	9.61	2	13.19	2	15.95	8	9.14	8
2pipe_2.ooo	925	8212	12.26	2	21.59	2	20.17	32	11.15	32
2pipe	861	6695	3.19	32	7.6	8	7.28	128	3.21	128
3pipe	2392	27.5k	72.09	32	165.75	8	163.2	512	70.95	512

Each $C(\pi, i)$ is then expressed in the CNF form using $i-1$ auxiliary variables $e_j = (x_j = x_j^\pi)$

$$\begin{aligned} C(\pi,i) =& (e_1 e_2 \dots e_{i-1} \Rightarrow (x_j \le x_j^\pi)) \\ =& (\bar{e}_1 + \bar{e}_2 + \dots + \bar{e}_{i-1} + \bar{x}_i + x_i^\pi). \end{aligned} \tag{4}$$

Due to clauses of growing size, CNF expressions for each $LL(\pi)$ have $\Theta(n^2)$ literals, which may be prohibitively expensive even for one permutation π with say, 9000 variables (see Table IV). Also, $LL(\pi)$ for different π may contain redundant clauses. To prune redundant clauses, the authors propose the concept of a symmetry tree, but it does not always prevent redundant clauses and is itself not always prunable to polynomial size [19].[5]

The need for more efficient, and also partial symmetry-breaking has been understood for some time [19], [39], [41], but no satisfactory generic approaches have been proposed that can be fully automated. In a recent paper [39], Luks and Roy show that, even for an Abelian (commutative) symmetry group and a given ordering of variables, full lex-leader symmetry-breaking predicates can be exponentially large. This drawback can be avoided by reordering variables, which allows polynomial-sized full lex-leader symmetry-breaking predicates for Abelian symmetry groups. However, the construction in [39] is not practical and is rather used for an existence proof. Also, it does not address non-Abelian groups.

[5]In the special case of the symmetry group S_n, according to [19], the symmetry-breaking predicate produced using a symmetry tree has size $\Theta(n^2)$. Techniques proposed in our paper generate a linear-sized predicate.

B. Using Symmetry Generators

In this paper, we explore partial symmetry breaking, i.e., we do not require that symmetry-breaking predicates be satisfied by lex-leaders only (but we do require that all lex-leaders satisfy symmetry-breaking predicates). Like other authors, we compute symmetry-breaking clauses on a per-symmetry basis, but consider only irredundant sets of symmetry generators (returned by graph automorphism programs), and not the entire symmetry group G. This idea was used in [19] in the context of pigeonhole instances. By breaking generator symmetries only, one does not necessarily break all symmetries. However, one can often achieve significant pruning because an irredundant set of generators contains "maximally independent" symmetries—none of them can be expressed in terms of others. The following example suggested to us by Eugene Goldberg of Cadence Berkeley Labs shows that symmetry-breaking by generators is not complete in some cases.

Consider a formula with four Boolean variables x_1, x_2, x_3, and x_4 that can be permuted arbitrarily, e.g., $(x_1+x_2+x_3+x_4)$. The symmetry group, S_4, can be given by the two generators $g_1 = (12)$ and $g_2 = (1234)$. Assume that, in each equivalence class of truth assignments under those symmetries, we select the lexicographically smallest element with respect to the original order of variables, i.e., x_1 is the most significant bit. The Boolean cube is split into five equivalence classes by the

TABLE IV
CHAFF RUN-TIME ON ORIGINAL SAT INSTANCES IS COMPARED TO THE COMBINED RUN-TIME OF SYMMETRY EXTRACTION AND CHAFF ON INSTANCES WITH SYMMETRY-BREAKING CLAUSES ADDED. THE RIGHTMOST COLUMN ALSO SHOWS PURE SEARCH SPEED-UP (THAT DOES NOT TAKE SYMMETRY EXTRACTION INTO ACCOUNT). THE FULL NAME OF BENCHMARK 2DLX_CA_MC IS 2DLX_CA_MC_EX_BP_F. THE NUMBERS OF SYMMETRY GENERATORS AND MAX CYCLES USED PER GENERATOR (10 OR ALL) ARE SHOWN. THE BENCHMARKS WE GENERATED FOR THESE EXPERIMENTS ARE AVAILABLE AT HTTP://GIGASCALE.ORG/BOOKSHELF/SLOTS/SATBENCH

Instance	Satis-fiable?	#vars and #clauses	Plain CHAFF sec	Time -out %	Symmetries					Speed-up: total / search only
					Extraction sec	Number of	#generators cycles		CHAFF sec	
hole07	UNS	56;204	0.37	0%	0.1	2.03e8	all	13	0.01	3.32; 36.50
hole08	UNS	72;297	1.27	0%	0.07	1.46e10	all	15	0.01	15.22; 94.15
hole09	UNS	90;415	3.79	0%	0.1	1.32e12	all	17	0.02	32.0; 204.97
hole10	UNS	110;561	22.44	0%	0.15	1.45e14	all	19	0.02	132; 1122
hole11	UNS	132;738	212.73	0%	0.18	1.91e16	all	21	0.03	1.23k; 7.09k
hole12	UNS	156;949	>1000	100%	0.24	2.98e18	all	23	0.04	— ; —
Urq3_5	UNS	46;470	>1000	100%	0.48	5.37e8	all	29	1.05	—;—
grout3.3-01	SAT	864;7592	19.01	0%	4.79	8.71e9	10	26	0.67	3.48; 28.37
grout3.3-03	SAT	960;9156	44.35	0%	8.94	6.97e10	10	29	0.40	4.75; 110.9
grout3.3-04	SAT	912;8356	19.36	0%	6.81	2.61e10	10	27	0.36	2.70; 53.79
grout3.3-08	SAT	912;8356	21.30	0%	7.14	3.48e10	10	28	0.67	2.73; 31.80
grout3.3-10	SAT	1056;10.8k	28.18	0%	10.65	3.48e10	10	28	0.85	2.45; 33.15
chnl10x11	UNS	220;1122	22.17	0%	0.45	4.20e28	all	39	0.11	39.91; 210.1
chnl10x12	UNS	240;1344	81.88	0%	0.61	6.04e30	all	41	0.12	111.6; 663.0
chnl10x13	UNS	300;2130	657.61	25%	1.28	4.50e37	all	47	0.17	454.8; 3.96k
chnl11x12	UNS	264;1476	207.37	0%	0.75	7.31e32	all	43	0.15	231.3; 1.41k
chnl11x13	UNS	286;1742	788.32	20%	1.08	1.24e35	all	45	0.16	633.5; 4.79k
chnl11x20	UNS	440;4220	>1000	100%	4.4	1.89e52	all	59	0.31	—; —
fpga10.08	SAT	120;448	7.56	0%	0.63	6.00e71	all	62	0.05	11.15; 157.6
fpga10.09	SAT	135;549	3.80	0%	0.88	6.33e77	all	68	0.03	4.16; 113.4
fpga12.11	SAT	198;968	694.00	50%	3.76	7.18e77	all	95	0.06	181.6; 11.3k
fpga12.12	SAT	216;1128	80.20	0%	5.31	7.44e77	all	104	0.13	14.74; 616.9
fpga12.08	SAT	144;560	246.70	10%	1.23	8.41e77	all	72	0.08	188.4; 3.10k
fpga12.09	SAT	162;684	885.00	80%	1.7	2.25e77	all	79	0.05	504.6; 16.4k
fpga13.09	SAT	176;759	550.00	85%	2.57	2.56e77	all	84	0.06	208.8; 8594
fpga13.10	SAT	195;905	>1000	100%	4.04	5.76e77	all	93	0.08	— ; —
fpga13.12	SAT	234;1242	>1000	100%	6.9	8.85e77	all	110	0.08	— ; —
2dlx_ca_mc*	UNS	3250;24.6k	6.54	0%	38.36	4	10	2	6.30	0.15; 1.04
2pipe	UNS	892; 6695	2.08	0%	10.74	128	10	7	1.56	0.17; 1.33
2pipe_1_ooo	UNS	834; 7026	2.55	0%	9.37	8	10	3	1.80	0.23; 1.41
2pipe_2_ooo	UNS	925; 8213	3.43	0%	11.14	32	10	5	2.82	0.25; 1.22
3pipe	UNS	2468;27.5k	36.44	0%	463.57	512	10	9	19.65	0.08; 1.85
4pipe	UNS	5237;80.2k	337.61	0%	>1000	—	—	—	—	—; —
5pipe	UNS	9471;195k	325.92	0%	>1000	—	—	—	—	—; —

action of S_4 because the number of 1's in truth assignments is invariant under permutational symmetries. In particular, the equivalence class of the truth assignment 0101 has six elements, and the smallest element is 0011. However, if we build symmetry-breaking predicates using g_1 and g_2 only, 0101 will satisfy them because $g_1(0101) = 1001 > 0101$ and $g_2(0101) = 1010 > 0101$. Thus, such symmetry-breaking predicates select more than one representative from some equivalence classes. Moreover, conjoining symmetry-breaking predicates for *powers* of generators does not help in this case because $g_1^2 = (\cdot)$, $g_2^4 = (\cdot)$, $g_2^2(0101) = 0101$, and $g_2^3(0101) = 1010 > 0101$.

Interestingly, for the symmetry group S_4, GAP/GRAPE/NAUTY do not produce the two generators used in the above example. They produce the following set of three generators: (12), (23), and (34). Our construction proposed below generates the symmetry-breaking clauses $(x_1 \leq x_2)$, $(x_2 \leq x_3)$, and $(x_3 \leq x_4)$, which admit only five truth assignments: 0000, 0001, 0011, 0111, and 1111—one from each equivalence class under S_4. This analysis shows that the particular choice of irredundant generating sets is important for symmetry-breaking. In our experience, GAP/GRAPE/NAUTY often produce "lucky" sets of generators that lead to fuller symmetry breaking. Our future research will attempt to explain why that is happening.

As shown by our experiments in Section V below, symmetry breaking by generators offers an attractive tradeoff between effective pruning and small overhead. However, we would

like to articulate an important pitfall in this direction. Firstly, adding symmetry-breaking predicates should not change the satisfiability of the original CNF instance. This is ensured by the fact that symmetry-breaking predicates are satisfied by at least one truth assignment from each class of symmetric truth assignments. The lex-leader predicates described above are satisfied by lexicographically smallest truth assignments because all $\mathrm{LL}(\pi)$ are. The pitfall lies in the possibility to conjoin symmetry-breaking predicates that are satisfied by nonlex-leader representatives of classes of symmetric truth assignments. A conjunction of such predicates may be unsatisfiable and, thus, unusable as a symmetry-breaking predicate. Therefore, in this paper, we adhere to lex-leader predicates.

C. Using Cycles of Permutations

Our construction is formulated in terms of cycles of a permutation. This is convenient because the output of graph automorphism programs is expressed in cycle notation. We observe that in overwhelmingly many instances all generators have two cycles only. Even in rare cases when three cycles were present, two cycles dominated by far. Another important observation about the output of graph automorphism programs is that collections of two cycles returned on the output are sorted according to the given variable ordering. Therefore, we can apply the Crawford construction in (2) and (3) to individual cycles and further optimize it for two cycles. In particular, for the variable swap (ab) the construction in [19] produces one additional variable and six symmetry-breaking clauses. Our construction below produces only one clause.

Single Cycles: First, observe that if the cycle (ab) is a symmetry, whenever there is a satisfying assignment with $a = 0$, $b = 1$, there should be a symmetric (equivalent) satisfying assignment with $a = 1$, $b = 0$ and other variables unchanged. To allow only the first assignment, we add the symmetry-breaking clause $(\bar{a} + b)$, which can also be interpreted as $(a \leq b)$. Similarly, to "break" a cycle of length three (abc), we add $(\bar{a}+b)(\bar{b}+c)$, i.e., $(a \leq b)(b \leq c)$. To make sure that the lexicographically smallest representatives of symmetric truth assignments satisfy our predicates, one has to choose an ordering of all variables at the beginning, and always use the $\leq$ sign consistently with that ordering. When $a = \bar{b}$, we get the cycle $(\bar{a}a)$ and it can be broken in two ways. In terms of the original CNF instance, the value of a can be fixed arbitrarily, and this can be expressed by a single one-literal symmetry-breaking clause: (a) or $(\bar{a})$. The construction in [19] does not address such phase-shift symmetries and never results in one-literal clauses. Our paper addresses arbitrary compositions of phase-shift and permutational symmetries.

In general, longer cycles require more complex symmetry-breaking clauses, but apparently one can always improve on the construction from [19]. A particular difficulty with cycles of length > 3 is that they cannot, in general, be ordered according to a given ordering of variables. For example, the cycle (1324) can be written as (3241), (2413), or (4132), but none of these representations are ordered. Therefore, we are not considering longer cycles in this paper (and they do not appear useful for symmetry breaking on our benchmarks).

Multiple Cycles: While single-literal symmetry-breaking clauses are most efficient (they reduce the solution space by 50%), they are associated with variables whose values do not affect satisfiability. After such variables are found and eliminated, other symmetries may remain. Indeed, we can produce symmetry-breaking clauses from any one two cycle or three-cycle of any symmetry. Yet, clauses of the form $(\bar{a} + b)$ achieve no pruning when $a = b$. A key idea in that case, similar to that in [19], is to process another cycle, but only if $a = b$. In fact, this is similar to (3), except that we now operate on cycles and do not need to involve *all* variables, which can dramatically reduce the size of symmetry-breaking clauses. Specifically, when building a symmetry-breaking predicate for the symmetry $(ab)(cd)(ef)\ldots$, we first add $(\bar{a} + b)$, then $(a = b) \Rightarrow (c \leq d)$, then $((a = b)(c = d)) \Rightarrow (e \leq f)$, etc. In the spirit of (4), we introduce one additional variable per cycle to indicate the equality of all variables in the cycle. A sample clause with new variables looks like $(\bar{x}_{a=b} + \bar{x}_{c=d} + \bar{e} + f)$. This construction is given only for permutations with two and three cycles.

Both (3) and our construction essentially perform a lexicographic comparison between the tested truth assignment and its symmetric image. The former operates on bits; the latter on cycles. In practice, this often leads to very large reductions in the number of generated clauses. As a result of the bitwise comparison, lexicographically smallest truth assignments are identified if single-bit comparisons are performed according to the global ordering of variables. However, in the context of cyclewise comparison, the situation is more complex. We only assert that a lexicographic comparison is performed when 1) each cycle is a two cycle; 2) each cycle is ordered according to the global ordering of variables; and 3) cycles are ordered lexicographically (which is equivalent to ordering them by the first element since they must be disjoint). Any chain of two cycles can be brought to this form by sorting.

Theorem 3.3.1: Consider an arbitrary single permutation consisting of two cycles only. Apply the proposed construction of symmetry-breaking predicates, including the sorting of cycles and elements within each cycle. All resulting CNF clauses are satisfied by lexicographically smallest representatives of classes of truth assignments that are symmetric under the given permutation. No other truth assignments satisfy all of those clauses.

Proof: Note that variables not involved in any cycles can be skipped during a lexicographic comparison of a truth assignment to its image under the given permutation. Our construction skips such variables. The rest of the proof employs induction on the number n_c of cycles. In the base case $n_c = 0$, the lexicographic comparison always returns true, and no clauses are generated. For an added cycle (ab) where a precedes b, we note that the clause $(\bar{a}+b)$, also known as $(a \leq b)$, lexicographically compares the partial assignments ab and ba. In other words, the test $(\bar{a}+b)$ checks that the value of a in the current truth assignment is $\leq$ to the value of b in the symmetric assignment. If the values are different, the overall comparison is finished. Otherwise, the comparison shifts to the least variable unseen before (which may be ordered before or after b) and its image under the permutation. This corresponds to considering the next two cycle. We would like to articulate that our construction does not

require variables in two cycles to be pairwise-adjacent in the variable ordering.

Since the square of the permutation is the identity, the classes of symmetric truth assignments consist of one or two elements only. The clauses we consider are satisfied by a given assignment if and only if (by construction) the image of this assignment is not lexicographically smaller than the assignment itself. Therefore, all clauses are satisfied by 1) one-element classes and 2) the smaller elements of all two-element classes. □

In our experiments, most generators returned by graph automorphism software consist of two cycles only. For rare benchmarks, some generators have small numbers of cycles of other lengths, typically three-cycles. It turns out that three-cycles can be ignored without violating the correctness of the symmetry-breaking procedure.

Theorem 3.3.2: Consider a single permutation having 1) cycles of length two, 2) cycles of odd lengths, and no other cycles. If the proposed construction of symmetry-breaking clauses is applied to two cycles only, the resulting clauses must be satisfied by all lex-leader truth assignments, and potentially other truth assignments.

Proof: Consider the product (or the least common multiple) p of all odd cycle lengths. The pth power of the given permutation has the same two cycles, but no other cycles. Since it is also a symmetry, Theorem 3.3.1 applies. Moreover, any lex-leader truth assignment with respect to the original permutation (i.e., cannot be improved by applying the permutation or its powers) is also a lex-leader with respect to the pth power. □

D. Further Improvements

In practice, the run-time for constructing symmetry-breaking clauses is often dwarfed by symmetry extraction run-time. Yet, with every cycle processed, we add larger and larger clauses. Large clauses that do not affect satisfiability rarely improve run-time of SAT solvers, so we optionally limit symmetry-breaking clauses to the first ten cycles of every symmetry. For the price of incomplete symmetry extraction, this technique considerably reduces the overhead of symmetry-breaking clauses. Based on Theorem 3.3.1, we make the following observation.

Observation 3.4.1: Consider a variant of the proposed construction of symmetry-breaking predicates (SBPs) for permutations with two cycles only. After cycles are sorted, only the first k cycles are considered and the remaining cycles ignored. The clauses produced by this reduced construction are all satisfied by lex-leader truth assignments, but other truth assignments may satisfy those clauses. Choosing two cycles at random may lead to inconsistent SBPs.

The reduced construction achieves less pruning than the full construction using all cycles, but its overhead is smaller. In our experiments, the reduced construction performed better. To further reduce overhead, a backtrack SAT solver can dynamically check for conditions of the form $((a = b)(c = d)\ldots(u = v))$. However, this paper discusses only preprocessing methods.

IV. Difficult SAT Instances

The *pigeonhole principle* asserts that $n+1$ pigeons cannot be assigned to n holes as long as 1) two pigeons are not assigned to the same hole and 2) every pigeon must be assigned to one hole. These constraints can be expressed in terms of $n(n+1)$ Boolean variables: x_{ij} is interpreted as the indicator of assignment of pigeon j to hole i. The first family of clauses consists of $n^2(n+1)/2$ mutual exclusions $(\overline{x_{ij_1}} + \overline{x_{ij_2}})$, $j_1 \neq j_2$. The second family consists of $n+1$ n-literal clauses $(\sum_{i=1}^{n} x_{ij})$—one for every pigeon j. The pigeonhole principle then asserts that those two families of clauses cannot be satisfied simultaneously. However, its easy proof by induction is beyond the capabilities of backtrack SAT-solvers that typically operate within the resolution proof system.

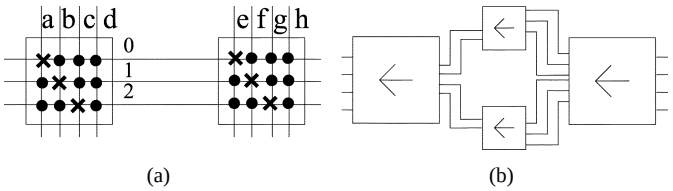


Fig. 4. Construction of difficult SAT instances. (a) Two switchboxes in common FPGA architectures and (b) similar N-by-M switchboxes are used to build hard satisfiable instances. Four connections are sought between a, b, c, and d and e, f, g, and h in (a). Crosses correspond to input connections mated to channels, and every solid dot indicates the absence of a link.

The pigeonhole instances hole-n described above are provably difficult for backtrack SAT solvers tied to resolution [7] and empirically difficult for the leading-edge implementation CHAFF as shown in Table IV. However, they are often treated as artificial in the EDA literature. Below, we derive equivalent instances $\texttt{chnl}[\texttt{N}] \times [\texttt{N}+1]$ from the domain of detailed routing for FPGAs and generalize them in two ways: $\texttt{chnl}[\texttt{N}] \times [\texttt{M}]$ (unsatisfiable) and $\texttt{FPGA}[\texttt{N}] \cdot [\texttt{M}]$ (satisfiable). We also give randomized constructions of difficult global routing instances grout.

A. FPGA Routing Instances

The pigeonhole principle is directly related to routing because it can be interpreted as the impossibility of routing $k+1$ connections through k channels. As one can imagine, trying to make m connections through n channels is typical for FPGA routing, and in some cases $m > k$. We encode such instances in terms of $m \times k$ FPGA switchboxes that mate m input connections to k channels. A switchbox can connect any given input to any one channel, but no two inputs can be connected to the same channel, and every input must be connected to some channel. The state of an FPGA switchbox is described by an $m \times k$ matrix of binary variables and, similar to the encoding of the pigeonhole principle above, is subject to two families of constraints. These constraints are violated if and only if there are fewer channels than inputs. We put two $m \times k$ switchboxes on both sides of a batch of k channels, which produces $2mk$ variables (see [46] for details of SAT formulations). Fig. 4(a), which illustrates our construction, shows two 4×3 FPGA switchboxes connected to three horizontal channels. Four connections are sought between 1) a, b, c, and d on the left and 2) e, f, g, and h on the right. Crosses represent input connections mated to channels, and every dot indicates the absence of a link. Empirical results in Table IV are shown for six routing configurations (chnl) in which one tries to route (a) 11, 12, or 13 connections through ten tracks, and (b) 12, 13, or 20 connections through 11 tracks. These instances are extremely difficult for the leading-edge SAT solver CHAFF [45] and also have many symmetries. They can appear as subin-

TABLE V
RUN-TIME OF THE BERKMIN (VERSION 56), SATZ, AND JERUSAT SOLVERS [11], [34], [29], ON SAMPLE SAT INSTANCES: ORIGINAL AND WITH SYMMETRY-BREAKING PREDICATES ADDED

Instance	BerkMin		Satz		JeruSAT	
	orig.	with SBPs	orig.	with SBPs	orig.	with SBPs
hole10	110.00	0.01	41.70	0.03	312.00	0.01
Urq3_5	>1000	0.29	>1000	2.56	201.00	0.95
groute3.3-03	5.50	0.60	>1000	46.10	10.20	0.83
chnl10_11	110.00	0.02	224.00	0.05	315.00	0.01
fpga24_23	252.00	0.20	>1000	27.50	>1000	4.27
3pipe	2.70	0.30	>1000	>1000	357.00	198.00

stances in larger routing instances, and such subinstances may be difficult to find.

From the benchmarking point of view, it is natural to expect *unsatisfiable* instances among the most difficult to solve. Indeed, randomized restarts used by CHAFF [45] typically allow it to avoid difficult regions of the search space and to quickly find satisfying solutions if they exist. However, our second construction is designed to create difficult *satisfiable* instances that trap even the best solvers in hopeless regions of their solution space for a long time before a satisfying solution can be found. The main idea is to create a satisfiable instance with a large number of hard-to-avoid unsatisfiable subinstances. If the number of unsatisfiable branches is much larger than the number of satisfiable branches, then random restart will keep on jumping from one unsatisfiable branch to another for a long time. Solvers without random restarts also will need to prove the unsatisfiability of many branches.

Our second construction produces routing a number of wires through four FPGA switchboxes of the type used in the first construction. The rightmost switchbox in the configuration in Fig. 4(b) has several redundant outgoing tracks that are divided into two channels. Each channel is connected to a smaller switchbox with an insufficient number of outgoing tracks. The two groups of tracks that leave the smaller switchboxes are connected to the leftmost switchbox. When routing connections through tracks right to left, connections must be split between switchboxes subject to the throughput constraints of switchboxes. However, to an SAT solver, the throughput constraints are obscured by the pigeonhole principle. SAT solvers first partition the connections between the two channels and backtrack from every partition that does not lead to a satisfying assignment. If the capacities of the two channels leading to the smaller switchboxes are greater than the throughput of those switchboxes, an overwhelming majority of partitions will lead to unsatisfiable pigeonhole instances. On average, at least several such instances must be solved before a good partition is found. Empirical results for these satisfiable instances (FPGA) in Table IV show that they are difficult for CHAFF. We observe that these instances become harder when the difference between the throughput of the small switchboxes and the capacities of the channels that lead to them is increased. This is consistent with our observations for the unsatisfiable `chnl` instances. Conceivably, some SAT-solvers may order variables related to the leftmost switchbox first and find satisfying assignments faster than CHAFF. This is consistent with our empirical data for the BerkMin solver [11] in Table V. However, the configuration of switchboxes in Fig. 4(b) can be further modified to generate more difficult benchmarks. Specifically, one can add three new switchboxes on the left which are copies of existing three switchboxes on the right. The overall configuration will then be symmetric about the vertical axis passing through the currently leftmost switchbox in Fig. 4(b).

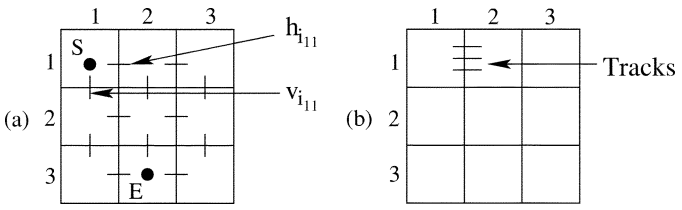


Fig. 5. Construction of difficult SAT instances (global routing).

B. Global Routing Instances

We propose a new construction of difficult randomized *satisfiable* instances unrelated to pigeonholes. They express routing two-pin connections in a grid with edge-capacity constraints. To ensure that an instance is satisfiable but difficult, we use *randomized flooding*. Namely, we create a routing configuration by adding shortest possible routes while unused routing resources (edge capacities) remain. Shortest routes are created by breadth-first-search between pairs of randomly chosen grid cells or, if that fails, by finding a maximal shortest route starting at a given grid cell with unused routing resources. After a routing configuration is created, routes are erased and their end-points are used to formulate an SAT instance.

Our SAT encoding of routing instances has two components. One deals with *route definition* and captures possible ways to route each connection. The other addresses *capacity constraints* and restricts the number of connections that can be routed across a grid cell boundary.

Route Definition Constraints: Routes are specified in terms of edges across cell boundaries in a grid. For each connection, we consider routing tracks across each cell boundary on the grid. In the SAT formulation, each track (for a given connection) is treated as a variable. Fig. 5(a) illustrates routing tracks in a 3×3 grid. Consider a two-terminal connection from S to E. Horizontal tracks for connection i are labeled $h_{i_{r,c}}$, where r and c are the row and column indices of the cell whose boundary the track crosses. Vertical tracks are labeled $v_{i_{r,c}}$. In Fig. 5(a), let the points marked S and E be the terminals of some two-terminal connection i. The SAT formulation proceeds as follows.

For every connection, we add groups of clauses corresponding to individual grid cells.

For each of the two terminals, we add a clause consisting of positive literals of variables of all tracks to which the terminal can connect. For example, we add the clause $(h_{i_{1,1}} + v_{i_{1,1}})$ for the terminal marked S in Fig. 5(a) because any route for this connection must pass through $h_{i_{1,1}}$ or $v_{i_{1,1}}$. In the general case, we also need to add [binary] mutual exclusion clauses ensuring that only one of the incident tracks is actually taken. For the terminal S, this produces only one clause $(\overline{h_{i_{1,1}}} + \overline{v_{i_{1,1}}})$. For the terminal E, this produces three clauses $(\overline{h_{i_{3,1}}} + \overline{v_{i_{2,2}}})(\overline{h_{i_{3,1}}} + \overline{h_{i_{3,2}}})(\overline{v_{i_{2,2}}} + \overline{h_{i_{3,2}}})$.

We now consider every grid cell other than the terminals. Either *none* or *two* of its boundary edges must be selected. This is enforced as follows. Observe that a given cell may have two, three, or four boundaries with tracks passing through them. Only two track variables, label them x_1 and x_2, are involved when "corner" grid cells are considered. In this case, we add clauses $(x_1 + \overline{x_2})(\overline{x_1} + x_2)$. In the case of three or four track variables ("border" grid cells or "internal" grid cells, respectively), we add two types of clauses. First, for every variable x_i, we add the constraint $(x_i \Rightarrow \sum_{j \neq i} x_j)$, which can be captured by one clause $(\overline{x_i} + \sum_{j \neq i} x_j)$ and says that if one boundary edge is selected, then another must be selected as well. The second type of clauses prohibits selecting three or four boundary edges. In the case of three variables x_1, x_2, and x_3, we add $(\overline{x_1} + \overline{x_2} + \overline{x_3})$. For four variables x_1, x_2, x_3, and x_4, we add

$$(\overline{x_1} + \overline{x_2} + \overline{x_3})(\overline{x_1} + \overline{x_2} + \overline{x_4})(\overline{x_1} + \overline{x_3} + \overline{x_4})(\overline{x_2} + \overline{x_3} + \overline{x_4}).$$

As an illustration, we apply this procedure to the grid cell (1,2) in Fig. 5(a) and produce

$$(\overline{h_{i_{1,1}}} + h_{i_{1,2}} + v_{i_{1,2}})(h_{i_{1,1}} + \overline{h_{i_{1,2}}} + v_{i_{1,2}})(h_{i_{1,1}} + h_{i_{1,2}} + \overline{v_{i_{1,2}}}) \cdot (\overline{h_{i_{1,1}}} + \overline{h_{i_{1,2}}} + \overline{v_{i_{1,2}}}).$$

The correctness of the general construction can be proven by the following argument. First, any given connection, interpreted as a truth assignment, satisfies those constraints. Now assume an arbitrary satisfying assignment and show that, topologically, it is a valid connection. Start at a terminal. Exactly one track must be taken toward a neighboring grid cell. If that cell is a terminal, we are done. Else, exactly one track must be taken to a cell not visited before. The same argument shows that if a partial route is not completed, it can be extended by one track. Since there is only a finite number of grid cells, the route must be completed sooner or later.

When the layout is not *obstructed*, the above construction can be applied to all grid cells in an arbitrary order. However, if some tracks are removed or if certain grid cells are not available for routing, some grid cells may be unreachable from the terminals. Since no routes can pass through unreachable grid cells, they can be ignored when an SAT instance is constructed. We perform this optimization by traversing grid cells by a breadth-first search. Once a terminal is enqueued, our algorithm enters a loop that dequeues one grid cell, marks it visited, adds relevant clauses, and enqueues unvisited adjacent grid cells. The algorithm finishes when the queue is empty. If the other terminal was not visited in the process, no routes connect the two terminals.

Capacity Constraints: Each edge of a grid cell boundary has a capacity associated with it to restrict the number of connections that can be routed through it. The capacity limits are intended to prevent routing congestion. If C is the capacity limit for an edge of a grid cell, we include C variables per edge for each connection. In other words, each connection can be routed through one of the C tracks across a cell boundary as shown in Fig. 5(b).

Consider two connections i and j. Consider horizontal tracks for each connection $h_{i_{r,c}}$, and $h_{j_{r,c}}$ for some row r and column c. Let $i_{r,c_1}, i_{r,c_2}, \ldots i_{r,c_C}$ and $j_{r,c_1}, j_{r,c_2}, \ldots j_{r,c_C}$ be the C extra variables introduced in the SAT formulation for the horizontal track in question. Then clearly, for any $i_{r,c_k}, 1 \leq k \leq C$, $i_{r,c_k} \Rightarrow h_{i_{r,c}}$, and also $h_{i_{r,c}} \Rightarrow (i_{r,c_1} + \cdots + i_{r,c_C})$. Clauses of this form are added to the SAT instance. Another restriction is that a route cannot pass through two tracks in the same channel (the edge of a grid cell), i.e., if for some k, $1 \leq k \leq C$, if i_{r,c_k} is true, then for all l, $1 \leq l \leq C, l \neq k$, $(i_{r,c_k} \Rightarrow \overline{i_{r,c_l}})$. These clauses are also added. Finally, two connections cannot be routed through the same track, i.e., for all k, $1 \leq k \leq C$, $(i_{r,c_k} \Rightarrow \overline{j_{r,c_k}})$ for all $j \neq i$, where j represents another connection.

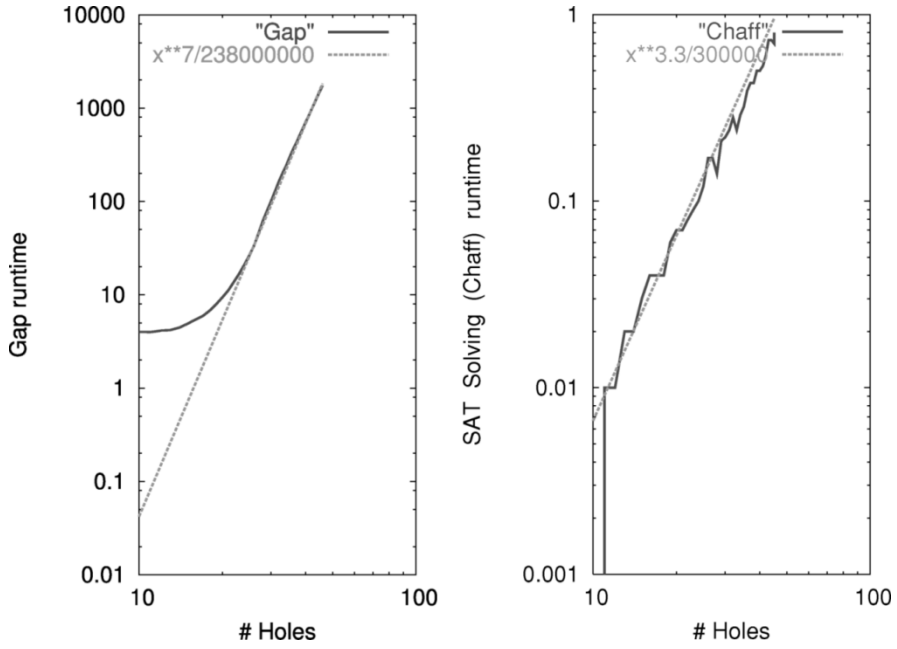
We created ten routing configurations by randomly flooding a 3×3 routing grid with connections subject to edge capacity constraints of 3. Then we applied the SAT encoding above. The difficulty of these randomly generated benchmarks varies, and we only report empirical results for the five most difficult instances (grout in Table IV).

V. Effect of Breaking Symmetries

Our computational experiments were performed on PCs with 1.2-GHz AMD Athlon processors and 1 GB of RAM. All codes were compiled with g++ 2.95.4 -O3 and ran on Debian Linux. The SAT solver used was CHAFF (MCHAFF version) [45]. In addition to the instances described in Section IV (`chnl` and FPGA) and (grout), Table IV lists six standard pigeonhole instances (hole), five families of artificially constructed randomized Urquhart benchmarks (Urq) [58], and seven recent benchmarks from the microprocessor verification domain [60].

CHAFF run-times in Table IV are averages of 20 independent starts because CHAFF uses randomization internally and results of different runs may vary significantly. All runs not completed in 1000 s were aborted and did not contribute to averages. The percent of time-outs is shown for each instance.

To extract symmetries from a CNF formula, we convert it into a colored graph as outlined in Section II. Those graphs are subsequently processed by the NAUTY program [42], [43]. For each run, the result is a list of permutation generators of the group of symmetries, specified by their cycles. For each SAT instance, Table IV lists NAUTY run-time in seconds excluding I/O, the total number of symmetries and the number of permutation generators. Those symmetry-extraction implementations are deterministic and are not affected by reordering of vertices in the input graph. For some benchmarks, we built symmetry-breaking clauses for only ten cycles per symmetry. The first ten cycles typically capture most of the speed-up provided by "breaking" a given symmetry. After new clauses were added, the preprocessed CNF instance was solved with CHAFF.



(a) (b)

Fig. 6. Plots of symmetry extraction time against $C_1 \cdot n^7$ and Chaff run-time against $C_2 \cdot n^{3.3}$ for pigeonhole instances (where n is the number of holes).

Table IV lists CHAFF run-times for each instance. Because CHAFF run-time on a given instance fluctuates from run to run, we report the averages of 20 independent runs for each instance. Preprocessed CNFs never timed out in our experiments.

The last column in Table IV shows the relative speed-up ratios due to the use of symmetry-breaking clauses. For a given CNF instance, the first number is the ratio of 1) the CHAFF run-time on original instance and 2) the total run-time of symmetry extraction and CHAFF on preprocessed instances. The second number is produced similarly, except that symmetry extraction run-time is ignored. This is the maximal possible speed-up if symmetries are found instantaneously or provided as domain-specific knowledge. We make the following observations.

1) The proposed SAT instances are only a fraction of the size of recent microprocessor verification benchmarks [60], but are more difficult to solve.
2) Some difficult SAT instances have astronomical numbers of symmetries; this includes the randomized Urq and grout benchmarks.
3) Symmetry-breaking clauses often speed-up the best available SAT solver CHAFF [45].
4) Symmetry-breaking clauses typically do not slow down CHAFF and often speed it up, even when few symmetries are present.
5) Either CHAFF or symmetry extraction may be a bottleneck.
6) Among the `chnl` instances, the hardest to solve was the routing of 20 connections through 11 tracks. Adding extra unrouted connections consistently increased difficulty. That is somewhat counterintuitive.

Not to limit our results to a single SAT-solver (CHAFF), we ran similar experiments with the BerkMin (version 56), Satz, and Jerusat solvers [11], [29], [34]. Representative results are shown in Table V where solver run-times are compared with and without symmetry-breaking predicates added. BerkMin solves the grout, FPGA, and microprocessor verification benchmark sets faster than CHAFF, but other benchmark sets are harder for BerkMin. JeruSAT solves the Urq benchmarks faster than both BerkMin and CHAFF, and is also faster than CHAFF on the grout instances. Satz is slower than all three other solvers on these benchmarks. Symmetry-breaking reduces run-time in most cases, for all solvers. In similar experiments with GRASP [50], all of our benchmarks are solved faster with the help of symmetry-breaking predicates, even if symmetry-extraction time is charged for.

Additionally, to support our claim that some families of SAT instances can be solved in polynomial time with symmetry breaking, we present the data in Fig. 6, which shows run-times for symmetry extraction (GAP) and SAT solving (CHAFF) on instances of the pigeonhole problem, plotted against a polynomial function of the number of holes (scaled by a constant). Fig. 6(a) shows GAP run-times (solid line) against $C_1 \cdot n^7$ (dashed line), for some constant C_1. Fig. 6(b) shows CHAFF run-times (solid line) against $C_2 \cdot n^{3.3}$. (dashed line), where n is the number of holes in a particular instance of the pigeonhole problem. The figure indicates that both symmetry-extraction and SAT solving run-times appear to exhibit polynomial growth.

VI. Opportunistic Symmetry Extraction

The use of symmetry-breaking clauses does not require extracting *all* symmetries. In fact, an algorithm that does not guarantee extracting all symmetries may finish sooner. Some symmetries may be found using domain-specific knowledge, and then symmetry-breaking clauses can be added during the creation of SAT instances.

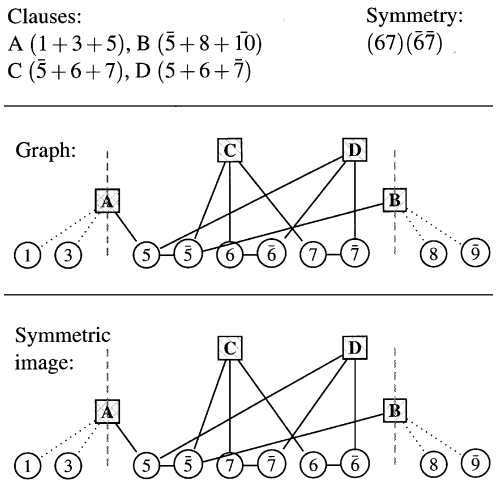


Fig. 7. Window-based opportunistic symmetry extraction for a CNF instance with ten variables and four clauses. Vertical dashed lines capture the window to which search for symmetries is limited. Each clause that includes literals from both within and beyond the window are represented by vertices of unique colors (dashed boxes). Symmetries are only allowed to permute vertices within the current window, therefore, vertices and edges beyond the current window are not included in the graph for window-based symmetry extraction. This reduces the size of graph automorphism problems.

A. Window-Based Symmetry Extraction

We observed that a variable would sometimes be symmetric to another variable connected by a clause (one hop) or through a chain of two clauses (two hops). When this is not true for all symmetries of a CNF formula, many symmetries may be composable from permutation generators of that kind. We, therefore, focus on "local" symmetries that permute small subsets of variables and fix all other variables.[6] We define the subsets by sliding a window of fixed size along a given linear ordering of the variables—either the original variable ordering of the CNF formula or the connectivity-based MINCE ordering [2]. For a window, we consider the left and right cuts, as in Fig. 7. To find symmetries local to a given window, the standard construction of colored graph is applied to clauses and literals that are entirely inside the window. Each *cut clause* is represented by a vertex of a unique color that is connected to literals inside the window. Vertices beyond the current window are ignored. To argue that the proposed construction is correct, i.e., does not add spurious symmetries, we consider the following *recoloring* process.

Definition 6.1.1: Given a colored graph and a subset of its vertices, change the color of each vertex into a unique color—one new color per vertex. This process is called recoloring *of a given set of vertices.* The following lemma shows how to restrict the set of symmetries of a colored graph. This can be done, e.g., with the purpose of accelerating symmetry extraction for the price of losing some symmetries.

Lemma 6.1.2: Given a colored graph G, consider an arbitrary recoloring of an arbitrarily-chosen subset of its vertices. Call the resulting graph G^R. Then the following claims hold.

a) *Every symmetry of G^R is a symmetry of G, and must map each recolored vertex to itself;*

b) *Symmetries of G^R form a subgroup in the group of symmetries G;*[7]

c) *The choice of new (unique) colors does not affect the symmetry group G^R.*

While reducing the number of symmetries can, in principle, be consistent with smaller symmetry extraction run-times, most graph automorphism programs are most sensitive to the number of vertices in the input graph rather than to the number of symmetries. The following lemma shows how to reduce the vertex set of the graph in the context of Lemma 6.1.2.

Lemma 6.1.3: Given a colored graph G, consider an arbitrary recoloring of an arbitrarily-chosen subset of its vertices. Call the recolored graph G^R. Consider a nonempty subset W of recolored vertices such that each of them is adjacent to recolored vertices only (if such a subset exists). Remove all vertices in W from G^R together with all incident edges. Then, the symmetries of the remaining colored graph G^R_W are in one-to-one correspondence with the symmetries of G^R, in fact the two groups of symmetries are isomorphic.

Proof: Every symmetry of G^R maps every vertex from W to itself by Lemma 6.1.2(a). Therefore, every such symmetry gives rise to a symmetry of G^R_W. Vice versa, every symmetry of G^R_W can be unambiguously extended to a symmetry of G^R by mapping every vertex from W to itself. This construction restores every symmetry of G^R mapped to a symmetry of G^R_W.

Lemma 6.1.3 reduces the number of vertices under the assumption that set W exists—the larger W, the greater the reduction. Constructively finding W remains an open problem.

Lemma 6.1.4: Given a colored graph G and an arbitrary edge-cut in it, pick one of the partitions and recolor all vertices in it. Then the set of vertices in that partition that are not incident to any edges in the cut can play the role of set W in Lemma 6.1.3.

Observe that colored graph G^R_W from Lemma 6.1.3 may still contain a large number of recolored vertices. This may be undesirable because the total number of vertices in G^R_W is limited by the scalability of available symmetry extraction software, and nontrivial symmetries of G^R_W do not involve recolored vertices. Indeed, recolored vertices are included into the vertex set of G^R_W, thus, potentially slowing down symmetry extraction programs or at least increasing memory usage.[8] Therefore, this construction can be improved by minimizing the number of vertices incident to cut edges, e.g., by minimizing the size of the cut itself.

Another concern about restricting symmetry extraction along the lines of Lemmas 6.1.2–6.1.4 is that one should apply it several times, with different sets of vertices recolored. This way more symmetries can be extracted. Indeed, if the size of G^R_W is limited by a constant, then the number of calls to symmetry extraction software should grow at least linearly so that every vertex in G be "given an opportunity" to map elsewhere.

The concerns mentioned above can be addressed in the context of window-based symmetry extraction. We first order CNF variables by representing the CNF as a hypergraph (clauses correspond to hyperedges) and heuristically finding a min-cut

[6]The complexity of such a restricted version of the graph automorphism problem was studied in [36].

[7]This subgroup is the *stabilizer* [26], [27] of the set of recolored vertices in the symmetry group of G.

[8]NAUTY maintains the input graph in a dense adjacency matrix.

TABLE VI
RESULTS FOR WINDOW-BASED SYMMETRY EXTRACTION. LABELING IS IDENTICAL TO THAT OF TABLE IV. TYPICALLY ALL OR A LARGE FRACTION OF ALL SYMMETRIES ARE DISCOVERED, COMPARED TO DATA IN TABLE IV

Instance	Satis-fiable?	#variables and #clauses	Plain CHAFF sec	Time -out %	Symmetries Extraction sec	# of	#generators cycles		CHAFF sec	Speed-up: total / search only
WINDOW-BASED SYMMETRY FINDING (1000 variables per window)										
2dlx_ca_mc*	UNS	3250;24.6K	6.54	0%	3.17	1	-	0	6.54	0.67; 1.00
2pipe	UNS	892; 6695	2.08	0%	10.47	128	10	7	1.30	0.18; 1.63
2pipe_1_ooo	UNS	834; 7026	2.55	0%	9.02	8	10	3	1.80	0.24; 1.41
2pipe_2_ooo	UNS	925; 8213	3.43	0%	11.09	32	10	5	2.80	0.25; 1.23
3pipe	UNS	2468;27.5K	36.44	0%	3.63	4	10	2	36.20	0.91; 1.01
4pipe	UNS	5237;80.2K	337.61	0%	9.32	2	10	1	334.0	0.98; 1.01
5pipe	UNS	9471;195K	325.92	0%	29.42	2	10	1	325	0.92; 1.00

linear arrangement of those vertices using recursive balanced bisection [2]. We then consider cuts along the resulting variable ordering, and those cuts are relatively small by construction. Note that cuts in Lemma 6.1.4 correspond to pairs of cuts in a given variable ordering as shown by vertical dashed lines in Fig. 7. Furthermore, in window-based symmetry extraction only clausal vertices can be recolored, therefore, min-cut linear arrangement naturally minimizes the number of recolored vertices.

We concatenate lists of permutation generators produced for different windows, consider the group generated by all those and use GAP [56] to produce an irredundant list of generators of this "global" group. Symmetry-breaking clauses are constructed from those generators. Observe that when applying symmetry extraction to a given window, we can only find symmetries that permute variables in that window only. Therefore, potentially more symmetries can be found if windows are allowed to overlap. On the other hand, if overlaps are allowed some symmetries may be found in multiple windows. Thus, producing symmetry-breaking clauses independently from each window and concatenating them may cause considerable redundancy. This is why we call GAP if windows are allowed to overlap. The tradeoff between run-time, incomplete symmetry extraction and redundancy among windows depends on their overlap. Similarly, the window size affects the tradeoff between run time and incomplete symmetry extraction. We observe good empirical performance with windows of size 1000. Results in Table VI show that our window-based technique found all or a significant portion of all symmetries for the microprocessor verification benchmarks [60] in a fraction of the run time spent by complete symmetry extraction If a randomized variable ordering is used, one could combine local permutation generators found for different orderings.

B. Improving SAT Formulations

One way to reduce the run-time of symmetry extraction is to learn how to extract (or predict) symmetries from domain-specific knowledge. Given the well-understood structure and symmetries of the hole, `chnl`, and FPGA benchmarks, we evaluated this approach on (randomized) grout benchmarks. We noticed that permuted variables in many cases correspond to neighboring tracks, e.g., if two connections are routed in parallel through several grid cells, there is considerable freedom (symmetry) in track assignment. To break this symmetry, we added clauses that preserve the relative order of tracks taken by every pair of connections routed through the same two edges of a grid cell. In other words, if one connection is routed through track 2 when entering the cell, and another connection is routed through track 3 when entering the cell, then the connections are allowed to leave the cell through tracks 2 and 3, respectively, 1 and 2 respectively, or 1 and 3 respectively. Such constraints speed-up CHAFF: each grout instance is now solved in *0.50–0.80 s versus 19–45 s*. More dramatic speed-ups are achieved for grout instances built with larger routing grids. Even if we apply symmetry extraction to modified instances, it completes much faster than on original instances because no symmetries are found. It may also be possible to add domain-specific symmetry-breaking clauses to SAT instances from [60] and improve CHAFF run-time according to results in Table IV.

VII. CONCLUSION

Our paper addresses solving difficult instances of Boolean (CNF) satisfiability that exhibit structural symmetries. While the utility of our approach on easy instances is not clear at this moment, the difficulty of domain-specific classes of CNF-SAT instances is often known, and adequate SAT algorithms can be chosen. Otherwise, several SAT solvers can be executed in parallel until one of them finishes. On a single processor, this may buy exponential speed-ups at the cost of a constant-factor slowdown. Therefore, our focus on difficult instances is well justified. Additionally, our experiments identify a number of difficult instances whose difficulty is apparently due to symmetries and the redundant search caused by them.

We describe an automated flow that finds symmetries in given CNF instances and uses them to speed up the SAT search. This flow includes symmetry extraction, preprocessing of given CNF instances, and an application of an existing state-of-art SAT solver. When compared to the SAT solver alone, applied to given CNF instances without preprocessing, our flow dramatically speeds up the solution of two well-known provably difficult benchmark families—pigeonhole problems and Urquhart benchmarks. Notably, methods proposed in a previous work [19] cannot find any nontrivial symmetries in Urquhart (Urq) benchmarks.

We offer constructions of realistic satisfiable and unsatisfiable SAT instances, arising in routing applications, that are unusually difficult for their size. Unlike most existing SAT bench-

marks, our benchmark families enable studies of the asymptotic performance of SAT solvers.

Since symmetry extraction is a bottleneck, we speed it up using opportunistic approaches. In one, we only look for symmetries that permute small groups of variables, determined by sliding a fixed-sized window along a given variable ordering. The second approach attempts to improve the construction of SAT instances by identifying symmetries in domain-specific terms. We find astronomically many symmetries in randomized Urq and grout benchmarks. This refutes a conventional-wisdom argument claiming that significant randomization destroys symmetries. We explain symmetries in grout benchmarks and break them using domain-specific knowledge.

Our proposed flow does not require source code modifications in SAT solvers and should work with most backtrack SAT solvers. We successfully validated our flow with CHAFF [45], BerkMin [11], Satz [34] and JeruSAT [29] solvers. Experiments performed with publicly available versions of WalkSAT [51] indicate that symmetry-breaking clauses do not improve runtimes, and even make them worse. This was observed by others and is the focus of ongoing work by Preswitch [49] as well as Kautz and Selman.

We stress that the proposed flow may not be useful on SAT benchmarks that 1) are easy or 2) do not have symmetries. Many difficult SAT instances do not have symmetries [17]. On the other hand, many DIMACS benchmarks [22] have large numbers of symmetries, but are easy and can be solved faster than their symmetries can be found by existing methods.

Our ongoing research seeks: 1) faster symmetry extraction, e.g., via incomplete algorithms; 2) finding [some] semantic symmetries that are not necessarily syntactic; 3) more efficient constructions of symmetry-breaking clauses; and 4) the use of partial/conditional symmetries. The latter were already shown useful in BDD-based model checking [24], SAT-solvers based on backtracking [14], [35] and more general constraint-satisfaction solvers [6].

ACKNOWLEDGMENT

The authors wish to thank S. V. Lokam of the Electrical Engineering and Computer Science Department, University of Michigan, Ann Arbor, for Lemma 2.3.3.

REFERENCES

[1] D. Achlioptas, C. P. Gomes, H. A. Kautz, and B. Selman, "Generating satisfiable problem instances," in *Proc. AAAI*, 2000, pp. 256–261.

[2] F. Aloul, I. Markov, and K. Sakallah, "Faster SAT and smaller BDD's via common structure," in *Proc. Int. Conf. Computer-Aided Design*, 2001, pp. 443–448.

[3] L. Babai and E. M. Luks, "Canonical labeling of graphs," in *Proc. Symp. Theory Comput.*, 1983, pp. 171–183.

[4] L. Babai, R. Beals, and P. Takácsi-Nagy, "Symmetry and complexity," in *Proc. Symp. Theory Comp.*, 1992, pp. 438–449.

[5] L. Babai, "Automorphism groups, isomorphism, reconstruction," in *Handbook of Combinatorics*, R. L. Graham, M. Grötschel, and L. Lovász, Eds. Cambridge, MA: MIT Press, 1995, vol. 2, ch. 27, pp. 1447–1541.

[6] R. Backofen and S. Will, "Excluding symmetries in constraint-based search," in *Proc. Int. Conf. Principles and Practice Constraint Program.*, vol. 1713, Lecture Notes in Computer Science, 1999, pp. 73–87.

[7] P. Beame, R. Karp, T. Pitassi, and M. Saks, "The efficiency of resolution and Davis-Putnam procedure," SIAM J. Comput., vol. 31, no. 4, pp. 1048–1075, to be published.

[8] B. Benhamou and L. Sais, "Tractability through symmetries in propositional calculus," *J. Automation Reasoning*, vol. 12, no. 1, pp. 89–102, 1994.

[9] C. Berman, "Circuit width, register allocation, and ordered binary decision diagrams," *IEEE Trans. Computer-Aided Design*, vol. 10, pp. 1059–1066, Aug. 1991.

[10] A. Bernasconi, V. Ciriani, F. Luccio, and L. Pagli, "Fast three-level logic minimization based on autosymmetry," in *Proc. Design Automation Conf.*, 2002, pp. 425–430.

[11] E. Goldberg and Y. Novikov, "BerkMin: A fast and robust SAT solver," in *Proc. Design, Automation Test Eur.*, 2002, pp. 142–149.

[12] D. Bosnacki, D. Dams, and L. Holenderski, "A heuristic for symmetry reductions with scalarsets," in *Proc. Int. Symp. Formal Methods for Increasing Software Productivity*, Lecture Notes in Computer Science, 2001, pp. 518–533.

[13] L. Brisoux, E. Gregoire, and L. Sais, "Improving backtrack search for SAT by means of redundancy," in *Proc. Int. Symp. Foundations Intell. Syst.*, Warsaw, Poland, 1999, pp. 301–309.

[14] C. A. Brown, L. Finkelstein, and P. W. Purdom, "Backtrack searching in the presence of symmetry," in *Proc. 6th Int. Conf. Applied Algebra, Algebraic Algorithms and Error Correcting Codes*, T. Mora, Ed., 1988, pp. 99–110.

[15] V. Chvatal and E. Szemeredi, "Many hard examples for resolution," *J. ACM*, vol. 35, no. 4, pp. 759–768, 1988.

[16] E. M. Clarke *et al.*, "Symmetry reductions in model checking," in *Proc. Int. Conf. Computer-Aided Verification*, A. J. Hu and M. Y. Vardi, Eds., 1998, pp. 159–171.

[17] S. A. Cook and D. G. Mitchell, "Finding hard instances of the satisfiability problem: A survey," in *Satisfiability Problem: Theory and Applications*, 1997, vol. 25, DIMACS Series in Discr. Math. and Theor. Comp. Sci, pp. 1–17.

[18] J. Crawford, "A theoretical analysis of reasoning by symmetry in first-order logic," in Proc. AAAI Workshop Tractable Reasoning, 10th Nat. Conf. Artif. Intell., San Jose, CA, 1992.

[19] J. Crawford, M. Ginsberg, E. Luks, and A. Roy, "Symmetry-breaking predicates for search problems," in *Proc. 5th Int. Conf. Principles Knowledge Representation Reasoning*, Cambridge, MA, 1996, pp. 148–159.

[20] M. Davis and H. Putnam, "A computing procedure for quantification theory," *J. ACM*, vol. 7, no. 3, pp. 201–215, 1960.

[21] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem proving," *J. ACM*, vol. 7, no. 5, pp. 394–397, 1962.

[22] DIMACS Boolean Satisfiability Challenge Benchmarks [Online]. Available: ftp://dimacs.rutgers.edu/pub/challenge/sat/benchmarks/cnf

[23] C. A. J. van Eijk, E. T. A. F. Jacobs, B. Mesman, and A. H. Timmer, "Identification and exploration of symmetries in DSP algorithms," in *Proc. Design Automation Test in Eur.*, Mar. 1999, pp. 602–608.

[24] E. A. Emerson and R. J. Trefler, "From asymmetry to full symmetry: New techniques for symmetry reduction in model checking," in *Prof. Conf. Correct Hardware Design and Verification Methods*, vol. 1703, Lecture Notes on Comp. Sci., 1999, pp. 142–156.

[25] E. Goldberg, "Testing satisfiability of CNF formulas by computing a stable set of points," in Proc. CADE, July 2002, pp. 161–180.

[26] M. Hall Jr., *The Theory of Groups*. New York: Macmillan, 1959.

[27] T. W. Hungerford, "Algebra," in *Graduate Texts in Mathematics*. New York: Springer-Verlag, 1973, vol. 73.

[28] C. N. Ip and D. L. Dill, "Better verification through symmetry," *Proc. Formal Methods Syst. Design*, vol. 9, no. 1–2, pp. 41–75, 1996.

[29] A. Nadel. JeruSAT Satisfiability Solver. [Online]. Available: http://www.geocities.com/alikn78/

[30] V. Kravets and K. Sakallah, "Generalized symmetries of boolean functions," in *Proc. Int. Conf. Computer-Aided Design*, 2000, pp. 526–532.

[31] ——, "Constructive library-aware synthesis using symmetries," in *Proc. Int. Conf. Design Automation Test Eur.*, 2001, pp. 208–213.

[32] B. Krishnamurthy, "Short proofs for tricky formulas," *Acta Informatica*, vol. 22, pp. 327–337, 1985.

[33] J. Köbler, U. Schöning, and J. Torán, "Graph isomorphism is low for PP," *Computat. Complex.*, vol. 2, no. 4, pp. 301–330, 1992.

[34] C. M. Li and Anbulagan, "Look-ahead versus look-back for satisfiability problems," in *Proc. 3rd Int. Conf. Principles Practice Constraint Program.*, LNCS 1330, Schloss Hagenberg, Austria, 1997, pp. 342–356.

[35] C. M. Li, B. Jurkowiak, and P. W. Purdom, "Integrating symmetry breaking into a DLL procedure," in *Intl. Symp. on Boolean Satisfiability (SAT)*, Cincinnatti, OH, 2002, pp. 149–155.

[36] A. Lozano and V. Raghavan, "On the Complexity of Moving Vertices in a Graph," Univ. Politéchnica de Catalunya, Barcelona, Spain, LSI-98–30-R, 1998.

[37] E. Luks, "Isomorphism of graphs of bounded valence can be tested in polynomial time," *Proc. IEEE Symp. Foundations Comput. Sci.*, pp. 42–49, 1980.
[38] ——, "Hypergraph isomorphism and structural equivalence of boolean functions," in *Proc. Symp. Theory Comput.*, 1999, pp. 652–658.
[39] E. Luks and A. Roy, "Symmetry breaking in constraint satisfaction," in Proc. Int. Conf. Artif. Intell. Math., Ft. Lauderdale, Florida, Jan 2–4, 2002.
[40] G. S. Manku, R. Hojati, and R. Brayton, "Structural symmetry and model checking," in *Proc. Int. Conf. Computer-Aided Verification*, 1998, pp. 159–171.
[41] I. McDonald and B. Smith, "Partial Symmetry Breaking,", APES-49–2002, 2002.
[42] B. D. McKay, "Practical graph isomorphism," in *Proc. Congressus Numerantium*, vol. 30, 1981, pp. 45–87.
[43] ——, "Nauty User's Guide," Comput. Sci. Dept., Australian Nat. Univ., Canberra, TR-CS-90–02, 1.5 ed., 1990.
[44] T. Miyazaki, "The complexity of Mckay's canonical labeling algorithm," in *Proc. Groups Computat. II , Workshop Groups Computat.*, DIMACS Series on Discrete Math. Theor. Comput. Sci., L. Finkelstein and W. M. Kantor, Eds., 1996.
[45] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "CHAFF: Engineering an efficient SAT solver," in *Proc. Design Automation Conf.*, 2001, pp. 530–535.
[46] G. Nam, F. Aloul, K. Sakallah, and R. Rutenbar, "A comparative study of two boolean formulations of FPGA detailed routing constraints," in *Proc. Int. Conf. Physical Design*, 2001, pp. 222–227.
[47] M. Prasad, P. Chong, and K. Keutzer, "Why is ATPG easy?," in *Proc. Design Automation Conf.*, 1999, pp. 22–28.
[48] M. Prasad, E. Goldberg, and R. Brayton, "Using problem symmetry in search based satisfiabiliy problems," in *Proc. Design Automation Test Eur.*, 2002, pp. 134–142.
[49] S. Preswitch, "Supersymmetric modeling for local search," in Proc. SymCon, Sept. 2002.
[50] J. P. M. Silva and K. A. Sakallah, "GRASP: A new search algorithm for satisfiability," *IEEE Trans. Comput.*, vol. 48, pp. 506–521, May 1999.
[51] B. Selman, H. A. Kautz, and B. Cohen, "Noise strategies for improving local search," in *Proc. Nat. Conf. Artif. Intell.*, 1994, pp. 337–343.
[52] B. Selman, D. Mitchell, and H. Levesque, "Generating hard satisfiability problems," *Artif. Intell.*, vol. 81, no. 1–2, pp. 17–29, 1996.
[53] A. Seress, "An introduction to computational group theory," *Notices Amer. Math. Soc.*, vol. 44, no. 6, pp. 671–679, 1997.
[54] B. M. Smith, K. E. Petrie, and I. P. Gent, "Models and Symmetry Breaking for Peaceable Armies of Queens,", APES-50–2002, 2002.
[55] L. H. Soicher, "GRAPE: A system for computing with graphs and groups," in *Groups and Computation*, L. Finkelstein and W. M. Kantor, Eds., 1993, vol. 11, DIMACS Ser. in Discr. Math. Theor. Comp. Sci., pp. 287–291.
[56] E. L. Spitznagel, "Review of mathematical software, GAP," *Notices Amer. Math. Soc.*, vol. 41, no. 7, pp. 780–782, 1994.
[57] G. S. Tseitin, "On the complexity of derivation in propositional calculus," in *Studies in Constructive Mathematics and Mathematical Logic, Part 2*. New York–London: Consultants Bureau, 1968, pp. 115–125.
[58] A. Urquhart, "Hard examples for resolution," *J. ACM*, vol. 24, no. 1, pp. 209–219, 1987.
[59] ——, *The Symmetry Rule in Propositional Logic*, 1996.
[60] M. N. Velev and R. E. Bryant, "Effective use of boolean satisfiability procedures in the formal verification of superscalar and VLIW microprocessors," in *Proc. Design Automation Conf.*, 2001, pp. 226–231.

Fadi A. Aloul (S'02) received the B.S. degree in electrical engineering (*summa cum laude*) from Lawrence Technological University, Southfield, MI, in 1997 and the M.S. and Ph.D. degrees in computer science and engineering from the University of Michigan, Ann Arbor, in 1999 and 2003, respectively.

He is currently a Post-Doc Research Fellow at the University of Michigan. His research interests are in the areas of computer-aided design, verification, and Boolean satisfiability.

Dr. Aloul is currently the AV Chair of the 2003 International Workshop on Logic Synthesis (IWLS). He is also serving on the technical committee of the International Workshop on Logic Synthesis, the International Conference on Theory and Applications of Satisfiability Testing, and the International Workshop on Soft Constraints. He has received a number of awards, including the Agere/SRC research fellowship, GANN fellowship, and the LTU presidential scholarship.

Arathi Ramani received the B.S. degree in computer engineering, from Thadomal Shahani Engineering College, affiliated with the University of Mumbai, India, in 1999 and the M.S. degree in computer engineering in 2002 from the University of Michigan, Ann Arbor, where she is currently persuing the Ph.D. degree.

Her interests are in algorithms for combinatorial optimization and their applications to electronic design automation.

Igor L. Markov received the M.S. degree in mathematics and the Ph.D. degree in computer science from the University of California, Los Angeles (UCLA).

He is an Assistant Professor of Electrical Engineering and Computer Science at the University of Michigan, Ann Arbor. His interests are in quantum computing and in combinatorial optimization with applications to the design and verification of integrated circuits. His contributions include the Capo circuit placer and quantum circuit simulator QuIDDPro. He has co-authored more than 50 publications.

Prof. Markov is serving on the technical program committees at the these fora: Design, Automation, and Test in Europe Conference, the International Symposium on Physical Design, the International Conference on Computer-Aided Design, the Great Lakes Symposium on Very Large Scale Integration, the International Workshop on System-Level Interconnect Prediction, the International Workshop on Logic and Synthesis, and the International Workshop on Symmetries in Contraint-Satisfaction Problems in 2003. He received the Best Ph.D. Student Award from the Department of Computer Science, UCLA in 2000.

Karem A. Sakallah (S'76–M'81–SM'92–F'98) received the B.E. degree in electrical engineering from the American University of Beirut, Beirut, Lebanon, and the M.S.E.E. and Ph.D. degrees in electrical and computer engineering from Carnegie Mellon University, Pittsburgh, PA, in 1975, 1977, and 1981, respectively.

In 1981, he was a Visiting Assistant Professor in the Department of Electrical Engineering, Carnegie Mellon University. From 1982 to 1988, he was with the Semiconductor Engineering Computer-Aided Design Group at Digital Equipment Corporation, Hudson, MA, where he headed the Analysis and Simulation Advanced Development Team. Since September 1988, he has been a Professor of Electrical Engineering and Computer Science at the University of Michigan, Ann Arbor. From September 1994 to March 1995, he was with the Cadence Berkeley Laboratory, Berkeley, CA, on a six-month sabbatical leave. He has authored or co-authored more than 150 papers and has presented seminars and tutorials at many professional meetings and various industrial sites. His research interests include the area of computer-aided design with emphasis on simulation, timing verification and optimal clocking, logic and layout synthesis, Boolean satisfiability, and design verification.

Dr. Sakallah was an Associate Editor of the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS from 1995 to 1997, and has served on the Program Committees of the International Conference on Computer-Aided Design, Design Automation Conference, and the International Conference of Computer Design as well as and numerous other workshops. He is currently an Associate Editor of the IEEE TRANSACTIONS ON COMPUTERS. He is a Member of the Associate of Computing Machinery (ACM) and Sigma Xi.