Efficient Symmetry Breaking for Boolean Satisfiability

Fadi A. Aloul, *Member*, *IEEE*, Karem A. Sakallah, *Fellow*, *IEEE*, and Igor L. Markov, *Senior Member*, *IEEE*

Abstract—Identifying and breaking the symmetries of conjunctive normal form (CNF) formulae has been shown to lead to significant reductions in search times. Symmetries in the search space are broken by adding appropriate symmetry-breaking predicates (SBPs) to an SAT instance in CNF. The SBPs prune the search space by acting as a filter that confines the search to nonsymmetric regions of the space without affecting the satisfiability of the CNF formula. For symmetry breaking to be effective in practice, the computational overhead of generating and manipulating SBPs must be significantly less than the runtime savings they yield due to search space pruning. In this paper, we describe a more systematic and efficient construction of SBPs. In particular, we use the cycle structure of symmetry generators, which typically involve very few variables, to drastically reduce the size of SBPs. Furthermore, our new SBP construction grows linearly with the number of relevant variables as opposed to the previous quadratic constructions. Our empirical data suggest that these improvements reduce search runtimes by one to two orders of magnitude on a wide variety of benchmarks with symmetries.

Index Terms—Backtrack Search, clause learning, conjunctive normal form (CNF), graph automorphism, satisfiability (SAT), symmetries.

1 Introduction

Modern Boolean satisfiability (SAT) solvers, based on backtrack search, are now capable of attacking instances with thousands of variables and millions of clauses [22] and are being routinely deployed in a wide range of industrial applications [2], [4], [11], [14], [20]. Their success can be credited to a combination of recent algorithmic advances and carefully tuned implementations [3], [8], [10], [13], [18], [23]. Still, there are problem instances that remain beyond the reach of most SAT solvers.

One aspect of intractability is the presence of symmetry in the conjunctive normal form (CNF) of an SAT instance. Intuitively, the symmetry of a discrete object is a transformation, e.g., a permutation, of its components that leaves the object intact. The symmetries of a CNF formula are permutations of its literals (variables and their negations) that result in a reordering of its clauses (and the literals within clauses) without changing the formula itself. Such symmetries induce an equivalence relation on the set of variable assignments such that two assignments are equivalent if and only if the formula assumes the same truth value (either 0 or 1) at each of these assignments. A search algorithm that is oblivious to the existence of these symmetries may end up, wastefully, exploring a set of equivalent unsatisfying assignments before moving on to a more promising region of the search space. On the other hand, knowledge of the symmetries can be used to significantly prune the search space. Symmetries are studied in abstract algebra in terms of groups. We assume the reader to be familiar with the basics of group theory; in particular, we assume familiarity with permutation groups and their representation in terms of irredundant sets of generators. A good reference on the subject is [9].

The rest of the paper is organized into five sections. Section 2 provides a brief review of permutations and permutation groups. Section 3 describes pervious work on symmetry breaking for SAT. Our main contribution on efficient constructions of symmetry-breaking predicates is detailed in Section 4. These constructions are evaluated empirically in Section 5 and we end with conclusions in Section 6.

• *F.A. Aloul is with the Computer Engineering Department. American University of Sharjah, PO Box 26666, Sharjah, UAE. E-mail: faloul@aus.edu.*
• *K.A. Sakallah and I.L. Markov are with the Electrical Engineering and Computer Science Department, University of Michigan, 1301 Beal Avenue, Ann Arbor, MI 48109-2122. E-mail: {karem, imarkov}@umich.edu.*

Manuscript received 5 Jan. 2005; revised 31 Aug. 2005; accepted 15 Nov. 2005; published online 22 Mar. 2006.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0001-0105.

2 Notation and Preliminaries

We will be concerned with permutations on the literals of a set of n Boolean variables, $\{x_1, \ldots, x_n\}$, which we assume to be totally ordered according to $x_1 < x_2 < \ldots < x_n$. We use I_n to denote the set of integers between 1 and n inclusive and denote nonempty subsets of I_n by uppercase "index variables" I and J as appropriate. Given an index set I and an index $i \in I$, we define the "index selector" functions:

$$pred(i, I) = \{j \in I | j < i\}, \tag{1}$$

$$prev(i, I) = \max(pred(i, I)), \tag{2}$$

$$succ(i, I) = \{j \in I | j > i\}, \tag{3}$$

$$next(i, I) = \min(succ(i, I)), \quad (4)$$

where min and max return, respectively, the least and greatest element in the given index set. For completeness, we also let $\min(\emptyset) = n + 1$ and $\max(\emptyset) = 0$.

A permutation π of the set of $2n$ literals $L = \{x_1, x_1', \ldots, x_n, x_n'\}$ (where x_i' denotes the logical negation of x_i) is a function $\pi : L \to L$ that is both one-to-one and onto. We will denote that x_j is the image of x_i under π by writing $x_j = x_i^\pi$. To preserve Boolean consistency, whenever π maps x_i to x_j, it must simultaneously map x_i' to x_j'. Such implied mappings will be assumed whenever not explicitly specified. A permutation π is a *phase-shift* permutation if $x_i^\pi = x_i'$ for some $i \in I_n$, i.e., π maps some literal to its complement.

Permutations will be expressed either in tabular form or in cyclic notation. For example,

$$\pi = \begin{pmatrix} x_1 x_2 \ldots x_n \\ x_1^\pi x_2^\pi \ldots x_n^\pi \end{pmatrix} \quad (5)$$

denotes a permutation that maps x_1 to x_1^π, etc. The same permutation can be expressed as a set of disjoint cycles, such as,

$$\pi = (x_i, x_i^\pi, (x_i^\pi)^\pi, \ldots)(x_j, x_j^\pi, (x_j^\pi)^\pi, \ldots) \ldots . \quad (6)$$

Here, a cycle $(a, b, \ldots, z)$ is a shortcut for "a maps to b, b maps to c, . . ., and z maps to a." The length of a cycle is equal to the number of literals in it; we will refer to a cycle whose length is k as a k-cycle. We define the support of a permutation π, $supp(\pi)$, to be the set of indices appearing in its cyclic representation, i.e.,

$$supp(\pi) = \{i \in I_n | x_i^\pi \neq x_i\}. \quad (7)$$

The number of cycles in a permutation π will be denoted by $cycles(\pi)$. We also define $phase\text{-}shift(\pi)$ to be the index of the *smallest* variable (according to the assumed total ordering) that is mapped to its complement by π:

$$phase\text{-}shift(\pi) = \min\{i \in I_n | x_i^\pi = x_i'\}. \quad (8)$$

We should note that a phase-shift permutation must have one or more *phase-shift cycles*, i.e., length-2 cycles that have the form (x_i, x_i'). Finally, we define $ends(\pi)$ as follows:

$$ends(\pi) = \left\{ i \in I_n | i \begin{array}{c} \text{is the largest index of a variable} \\ \text{in a non-phase-shift cycle of } \pi \end{array} \right\}. \quad (9)$$

A *permutation group* G is a group whose elements are permutations of some finite set and whose binary operation is function composition, also referred to as permutation multiplication. The order of a group is the number of its elements. A *subgroup* H of a group G, denoted $H \leq G$, is a subset of G that is closed under the group's binary operation. The *cyclic subgroup* of $\pi \in G$, denoted by $<\pi>$, is the subgroup consisting of π and its integer powers:

$$<\pi> = \{\pi^i | (i \in Z)\} \quad (10)$$

and π is said to generate $<\pi>$. A set of permutations $\pi_1 \in G, \ldots, \pi_k \in G$ generates G if the subgroup resulting from taking all possible products of the integer powers of these permutations is equal to G. The permutations $\{x_1, \ldots, x_k\}$ are called *generators* of G. A set of generators is *irredundant* if it is not possible to express any of its permutations as a product of powers of its other permutations. A set of irredundant generators serves as an implicit representation of the group it generates and, in general, guarantees exponential compression in the size of the representation. Note that a set of irredundant generators is not a group since it is not closed under multiplication and taking inverse. In the sequel, a set of permutations G that is not necessarily closed will be indicated by placing a "hat" on the variable denoting the set, i.e., $\hat{G}$. Additionally, and with a slight abuse of notation, we will indicate that G is the group generated by $\hat{G}$ by writing $G = < \hat{G} >$.

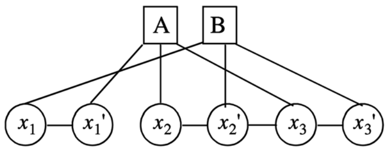


Fig. 1. Conversion of the CNF formula $(x_1' + x_2 + x_3)(x_1 + x_2' + x_3')(x_2' + x_3)$ to a graph for symmetry extraction. Different shapes correspond to different vertex colors.

3 Previous Work

The basic framework for utilizing the symmetries in a CNF instance to prune the search space explored by an SAT solver was laid out in [5]. This framework was extended later, in [1], to account for phase-shift symmetries, take advantage of the cycle structure of permutations, and consider only generators of the group of symmetries. In outline, the procedure consists of the following steps:

1. Convert a CNF formula φ to a colored graph whose symmetries are isomorphic to the symmetries of the formula. A simple construction represents every nonbinary clause by a vertex of color 2 and every variable by two vertices of color 1 (one for the positive and one for the negative literal) connected by Boolean consistency edges. Every literal in the CNF formula is then represented by a bipartite edge. Binary clauses are represented by connecting their literal vertices directly [1], [5]. An example is shown in Fig. 1.
2. Find the symmetries of the graph in terms of a set of irredundant generators $\hat{G} = \{\pi_1, \ldots, \pi_k\}$ using a suitable graph automorphism program [6], [12], [19].
3. Map the graph symmetries back to symmetries of the CNF formula.
4. Construct an appropriate *symmetry-breaking predicate* (SBP) ρ and conjoin it to the formula.
5. Solve $\varphi \wedge \rho$ using a suitable SAT solver [13].

Our concern in this paper is Step 4. Noting that the group of symmetries induces an equivalence relation on the set of assignments in the n-dimensional Boolean space, the basic idea is to construct a "filter" that picks out a single representative from each equivalence class. In particular, choosing the *lexicographically smallest representative*—according to the assumed total ordering on the variables—leads to the following *Lex-Leader SBP* [5]:

$$\pi = \begin{pmatrix} x_1x_2x_3x_4x_5x_6x_7x_8x_9x_{10} \\ x_4x_2x_8x_1x_5'x_3'x_7x_6'x_9x_{10} \end{pmatrix}$$

$$\pi = (x_1\,,x_4)(x_3\,,x_8\,,\,x_6')(x_5\,,\,x_5')$$

(a)

$$\begin{aligned}
&\mathrm{supp}(\pi) = \{1\,,3\,,4\,,5\,,6\,,8\}\\
&\text{phase-shift}(\pi) = 5\\
&\mathrm{succ}(\text{phase-shift}(\pi)\,,I_{10}) = \{6\,,7\,,8\,,9\,,10\}\\
&\mathrm{ends}(\pi) = \{4\,,8\}\\
&\mathrm{supp}(\pi)\backslash\mathrm{ends}(\pi) = \{1\,,3\,,5\,,6\}\\
&\mathrm{supp}(\pi)\backslash\mathrm{ends}(\pi)\backslash\mathrm{succ}(\text{phase-shift}(\pi)\,,I_{10}) = \{1\,,3\,,5\}
\end{aligned}$$

(b)

$$\mathrm{BP}(\pi\,,1) = x_1 \le x_4$$

$$\boxed{\mathrm{BP}(\pi\,,2) = (x_1 = x_4) \to (x_2 \le x_2)}$$

$$\mathrm{BP}(\pi\,,3) = (x_1 = x_4)(x_2 = x_2) \to (x_3 \le x_8)$$

$$\boxed{\mathrm{BP}(\pi\,,4) = (x_1 = x_4)(x_2 = x_2)(x_3 = x_8) \to (x_4 \le x_1)}$$

$$\mathrm{BP}(\pi\,,5) = (x_1 = x_4)(x_2 = x_2)(x_3 = x_8)(x_4 = x_1) \to (x_5 \le x_5')$$

$$\mathrm{BP}(\pi\,,6) = (x_1 = x_4)(x_2 = x_2)(x_3 = x_8)(x_4 = x_1)(x_5 = x_5') \to (x_6 \le x_3')$$

$$\boxed{\begin{aligned}
&\boxed{\mathrm{BP}(\pi\,,7) = (x_1 = x_4)(x_2 = x_2)(x_3 = x_8)(x_4 = x_1)(x_5 = x_5')(x_6 = x_3') \to (x_7 \le x_7)}\\
&\boxed{\mathrm{BP}(\pi\,,8) = (x_1 = x_4)(x_2 = x_2)(x_3 = x_8)(x_4 = x_1)(x_5 = x_5')(x_6 = x_3')(x_7 = x_7) \to (x_8 \le x_6')}\\
&\boxed{\mathrm{BP}(\pi\,,9) = (x_1 = x_4)(x_2 = x_2)(x_3 = x_8)(x_4 = x_1)(x_5 = x_5')(x_6 = x_3')(x_7 = x_7)(x_8 = x_6') \to (x_9 \le x_9)}\\
&\boxed{\begin{aligned}\mathrm{BP}(\pi\,,10) = {}&(x_1 = x_4)(x_2 = x_2)(x_3 = x_8)(x_4 = x_1)(x_5 = x_5')(x_6 = x_3')(x_7 = x_7)(x_8 = x_6')(x_9 = x_9) \to\\ &(x_{10} \le x_{10})\end{aligned}}
\end{aligned}}$$

(c)

$$\begin{aligned}
\mathrm{PP}(\pi) &= (p_1)(p_1 \to l_1p_3)(p_3 \to g_1 \to l_3p_5)(p_5 \to g_3 \to l_5)\\
&= (p_1)(p_1 \to (x_1 \le x_4)p_3)(p_3 \to (x_1 \ge x_4) \to (x_3 \le x_8)p_5)(p_5 \to (x_3 \ge x_8) \to x_5')
\end{aligned}$$

(d)

Fig. 2. Illustration of different formulations of the permutation predicate. (a) Permutation in tabular and cyclic notation. (b) Various index sets associated with permuation. (c) Bit predicates according to (10). BPs enclosed in boxes with square corners are tautologous because π maps the corresponding bits to themselves. BPs enclosed in boxes with rounded corners are tautologous because they correspond to cycle "ends." The BPs for bits 6 to 10 are tautologous because π maps bit 5 to its complement. (d) Linear formulation of the permutation predicate according to (18), based only on irredundant bits.

$$\rho^{LL}(<\hat{G}>) = \bigcap_{\pi\in<\hat{G}>} PP(\pi), \tag{11}$$

$$PP(\pi) = \bigcap_{i\in I} BP(\pi, i), \tag{12}$$

$$BP(\pi, i) = \left[\bigcap_{j\in pred(i,I)} (x_j = x_j^{\pi})\right] \to (x_i \le x_i^{\pi}), \tag{13}$$

where the index set I in (12) and (13) is equal to I_n. In these equations, the Lex-Leader SBP is expressed as a conjunction of *permutation predicates* (PPs), each of which is a conjunction of *bit predicates* (BPs).[1] Introducing n auxiliary "equality" variables, $e_i \equiv (x_i = x_i^{\pi})$, makes it possible to express the ith BP as an $(i+1)$-literal CNF clause. This leads to a CNF representation of the PP in (12) that has n clauses with a total literal count of $0.5(n^2 + 3n)$. Additionally, each of the introduced equality constraints yields four 3-literal clauses bringing the total CNF size of (12) to:

$$\begin{aligned} clauses(PP(\pi)) &= 5n\\ literals(PP(\pi)) &= 0.5(n^2 + 27n). \end{aligned} \tag{14}$$

In its present form, the lex-leader SBP in (11)-(13) can lead to an exponentially large CNF formula because the order of the symmetry group can be exponential in the number of variables. Thus, its value in pruning the search space is negated by the need of the SAT solver to process a much larger CNF formula. To remedy this problem, the authors of [5] suggested the construction of a symmetry tree to eliminate some redundant permutations. However, in the worst case, the number of symmetries in the tree remains

1. Note that $x \le y$ in the bit predicate mean "x implies y."

TABLE 1
Symmetry Statistics for Various Benchmark Families

1	2	3	4	5	6	7	8	9	10
Bench-mark Family	Instance	Instance Size			Symmetries				
					Total	Generators			
		#Variables	#Clauses	#Literals		Total	PS	|supp| Sum	S
Hole-n	hole07	56	204	448	2.03E+08	13	0	194	73.35%
	hole08	72	297	648	1.46E+10	15	0	254	76.48%
	hole09	90	415	900	1.32E+12	17	0	322	78.95%
	hole10	110	561	1210	1.45E+14	19	0	398	80.96%
	hole11	132	738	1584	1.91E+16	21	0	482	82.61%
	hole12	156	949	2028	2.98E+18	23	0	574	84.00%
Urq	Urq3_1	43	334	1872	6.71E+07	26	26	136	87.84%
	Urq3_4	36	220	1172	5.24E+05	19	19	105	84.65%
	Urq3_5	46	470	2912	5.37E+08	29	29	159	88.08%
	Urq3_9	37	236	1256	1.05E+06	20	20	129	82.57%
GRoute	grout3.3-03	960	9156	19258	6.97E+10	29	0	1440	94.83%
	grout3.3-04	912	8356	17612	2.61E+10	27	0	1200	95.13%
	grout3.3-10	1056	10862	22742	3.48E+10	28	0	1440	95.13%
FPGARoute	fpga10.08	120	448	1200	6.69E+11	22	0	512	80.61%
	fpga10.09	135	549	1485	1.50E+13	23	0	446	85.64%
	fpga12.08	144	560	1488	2.41E+13	24	0	624	81.94%
	fpga12.09	162	684	1836	5.42E+14	25	0	546	86.52%
	fpga12.11	198	968	2640	1.79E+18	29	0	678	88.19%
	fpga12.12	216	1128	3096	2.57E+20	32	0	960	86.11%
	fpga13.09	176	759	2031	3.79E+15	26	0	598	86.93%
	fpga13.10	195	905	2440	1.90E+17	28	0	668	87.77%
	fpga13.12	234	1242	3396	9.01E+20	32	0	812	89.16%
ChnlRoute	chnl10.11	220	1122	2420	4.20E+28	39	0	1016	88.16%
	chnl10.12	240	1344	2880	6.04E+30	41	0	1112	88.70%
	chnl10.13	260	1586	3380	1.02E+33	43	0	1208	89.19%
	chnl11.12	264	1476	3168	7.31E+32	43	0	1228	89.18%
	chnl11.13	286	1742	3718	1.24E+35	45	0	1334	89.63%
	chnl11.20	440	4220	8800	1.89E+52	59	0	2076	92.00%
XOR	x1_16	46	122	364	1.31E+05	17	15	127	83.76%
	x1_24	70	186	556	1.68E+07	24	24	280	83.33%
	x1_32	94	250	748	4.29E+09	32	32	338	88.76%
2pipe	2pipe_1.ooo	834	7026	19768	8.00E+00	3	2	724	71.06%
	2pipe_2.ooo	925	8212	23161	3.20E+01	5	4	888	80.80%
	2pipe	861	6695	18637	1.28E+02	7	4	880	85.40%
	Total	9826	74022	180854	1.89E+52	885	175	23888	86%

*PS: # phase-shift generators; |supp| Sum: sum of support of generators; S: sparsity of generators (1 - |supp|/(Vars*Gen)).*

exponential. Empirical evidence in [1] showed that full symmetry breaking, i.e., insuring that the SBP selects only the lex-leader from each equivalence class, is not necessary to obtain significant pruning of the search space. An SBP that breaks some, but not necessarily all, of the symmetries of the formula can, in fact, provide a much better space/time trade-off during the search. This is accomplished by replacing the group of symmetries in (11) by a suitable, and much smaller, set of permutations $\hat{H} \subseteq < \hat{G} >$:

$$\rho^{LL}(\hat{H}) = \bigcap_{\pi \in \hat{H}} PP(\pi). \tag{15}$$

In particular, the approach in [1] advocated the use of the set of generators $\hat{G}$ returned by the graph automorphism program in Step 2.

4 Efficient Formulation of Permutation Predicate

Even when only a small number of permutations is used in constructing an SBP, as in (15), the corresponding CNF formula may still be too large because each PP requires a CNF formula whose size is quadratic in the number of variables n. In this section, we introduce two refinements that lead to much smaller PPs. The first refinement utilizes the *cycle structure* of a permutation to eliminate redundant bit predicates and can be viewed as replacing n in (14) by a much smaller number m and represents a more comprehensive and systematic treatment of cycles than that in [1]. The second refinement takes advantage of the *recursive bit-by-bit* structure in (13) to yield a CNF formula whose size is

TABLE 2
Size Comparisons of Three SBP Constructions Based on Group Generators

11	12	13	14	15	16	17	18	19	20	21
Bench-mark Family	Instance	Total SBP Sizes								
		All Bits			Irredundant Bits					
		Quadratic Construction						Linear Construction		
		#Extra Var	#Clauses	#Literals	#Extra Var	#Clauses	#Literals	#Extra Var	#Clauses	#Literals
Hole-n	hole07	728	3640	30212	97	485	1673	97	362	1215
	hole08	1080	5400	53460	127	635	2254	127	478	1613
	hole09	1530	7650	89505	161	805	2938	161	610	2067
	hole10	2090	10450	143165	199	995	3731	199	758	2577
	hole11	2772	13860	220374	241	1205	4639	241	922	3143
	hole12	3588	17940	328302	287	1435	5668	287	1102	3765
Urq	Urq3_1	1118	5590	39130	0	26	26	0	26	26
	Urq3_4	684	3420	21546	0	19	19	0	19	19
	Urq3_5	1334	6670	48691	0	29	29	0	29	29
	Urq3_9	740	3700	23680	0	20	20	0	20	20
GRoute	grout3.3-03	27840	139200	13739040	720	3600	20280	720	2822	9761
	grout3.3-04	24624	123120	11560968	600	3000	15888	600	2346	8103
	grout3.3-10	29568	147840	16011072	720	3600	20136	720	2824	9772
FPGARoute	fpga10.08	2640	13200	194040	256	1280	6257	256	980	3342
	fpga10.09	3105	15525	251505	223	1115	4228	223	846	2869
	fpga12.08	3456	17280	295488	312	1560	8136	312	1200	4104
	fpga12.09	4050	20250	382725	273	1365	5300	273	1042	3547
	fpga12.11	5742	28710	645975	339	1695	6821	339	1298	4427
	fpga12.12	6912	34560	839808	480	2400	14904	480	1856	6368
	fpga13.09	4576	22880	464464	299	1495	5875	299	1144	3900
	fpga13.10	5460	27300	606060	334	1670	6677	334	1280	4368
	fpga13.12	7488	37440	977184	406	2030	8405	406	1560	5332
ChnlRoute	chnl10.11	8580	42900	1059630	508	2540	14997	508	1954	6683
	chnl10.12	9840	49200	1313640	556	2780	17102	556	2142	7333
	chnl10.13	11180	55900	1604330	604	3020	19325	604	2330	7983
	chnl11.12	11352	56760	1651716	614	3070	19772	614	2370	8123
	chnl11.13	12870	64350	2014155	667	3335	22371	667	2578	8843
	chnl11.20	25960	129800	6061660	1038	5190	44512	1038	4034	13883
XOR	x1_16	736	3680	26864	1	20	29	1	17	18
	x1_24	1680	8400	81480	0	24	24	0	24	24
	x1_32	3008	15040	181984	0	32	32	0	32	32
2pipe	2pipe_1.ooo	2502	12510	1077111	319	1597	54556	319	1274	4445
	2pipe_2.ooo	4625	23125	2201500	358	1794	67501	358	1430	4981
	2pipe	6027	30135	2675988	354	1774	64651	354	1410	4903
	Total	239K	1M	67M	11093	55640	468776	11093	43119	147618

Extra Vars: # of additional variables used in SBPs.

linear, rather than quadratic, in m. Fig. 2 provides an example illustrating these refinements.

4.1 Elimination of Redundant BPs

Careful analysis of (13) reveals three cases in which a BP is tautologous and, hence, redundant. The first corresponds to bits that are mapped to themselves by the permutation, i.e., $x_i^\pi = x_i$. This makes the consequent of the implication in (13), and, hence, the whole bit predicate, unconditionally true. Removal of such BPs is easily accomplished by setting the index set I in (12) and (13) to $supp(\pi)$ rather than I_n. For sparse permutations, i.e., permutations for which $|supp(\pi)| << n$, this change alone can account for most of the reduction in the CNF size of the PP.

The second case corresponds to the BP of the last bit in each cycle of π. "Last" here refers to the assumed total ordering on the variables. Assume a cycle involving the variables $\{x_j | j \in J\}$ for some index set J and let $i = \max(J)$. Then,

$$\left[\bigcap_{j \in J \setminus \{i\}} (x_j = x_j^\pi)\right] \rightarrow (x_i \leq x_i^\pi) = 1, \tag{16}$$

causing the corresponding bit predicate $BP(\pi, i)$ to be tautologous. Elimination of these BPs is accomplished by restricting the index set I in (12) and (13) further to just

TABLE 3
Comparison of Search Runtimes for Various Choices of SBP Constructions and Symmetries to Break

22	23	24	25	26	27	28	29	30	31	32	33
Benchmark Family	Instance	# Generators	# Generators & their compositions	Time to find symmetries (sec)	Time to solve orig. instance (sec)	Time to solve instances and SBPs (sec)				Speedup	
						Generators only			Generators & their compositions	$\frac{\text{col 28}}{\text{col 26 + col 30}}$	$\frac{\text{col 27}}{\text{col 26 + col 30}}$
						All Bits	Irredundant Bits				
						Quadratic construction		Linear construction			
Hole-n	hole07	13	102	0.00	0.03	0.03	0.01	0.01	0.01	3	3
	hole08	15	133	0.00	0.15	0.17	0.01	0.01	0.01	17	15
	hole09	17	168	0.01	0.97	0.30	0.01	0.01	0.01	15	49
	hole10	19	207	0.02	14.4	2.87	0.01	0.01	0.01	96	480
	hole11	21	250	0.02	133	9.04	0.01	0.01	0.02	301	4440
	hole12	23	297	0.02	>1000	6.90	0.01	0.01	0.03	230	>33333
Urq	Urq3_1	26	26	0.02	159	0.20	0.04	0.04	0.01	3	2650
	Urq3_4	19	19	0.01	0.13	0.02	0.01	0.01	0.01	1	7
	Urq3_5	29	29	0.04	>1000	0.62	0.89	0.88	0.01	1	>1087
	Urq3_9	20	20	0.01	5.14	0.01	0.01	0.01	0.01	1	257
GRoute	grout3.3-03	29	447	0.51	40.5	408	1.15	0.37	3.08	464	46
	grout3.3-04	27	391	0.46	1.12	199	0.32	0.03	1.89	262	1
	grout3.3-10	28	418	0.55	808	331	4.38	0.52	0.67	309	755
FPGARoute	fpga10.08	22	269	0.02	24.6	0.38	0.02	0.01	0.04	13	820
	fpga10.09	23	294	0.03	150	1.43	0.01	0.01	0.02	36	3750
	fpga12.08	24	318	0.03	389	1.36	0.01	0.01	0.10	34	9725
	fpga12.09	25	345	0.04	>1000	2.39	0.01	0.01	0.04	48	>20000
	fpga12.11	29	459	0.07	>1000	4.98	0.02	0.01	0.10	62	>12500
	fpga12.12	32	554	0.08	989	3.27	0.01	0.01	0.22	36	10989
	fpga13.09	26	372	0.05	>1000	3.75	0.02	0.02	0.05	54	>14286
	fpga13.10	28	429	0.06	>1000	8.48	0.02	0.01	0.10	121	>14286
	fpga13.12	32	555	0.10	>1000	20.8	0.02	0.01	0.14	189	>9091
ChnlRoute	chnl10.11	39	814	0.06	14.5	27.8	0.04	0.01	0.07	396	207
	chnl10.12	41	897	0.08	14.7	0.75	0.05	0.01	0.08	8	163
	chnl10.13	43	984	0.09	15.4	44.1	0.05	0.01	0.09	442	154
	chnl11.12	43	984	0.09	133	2.34	0.07	0.01	0.11	23	1334
	chnl11.13	45	1075	0.11	243	>1000	0.08	0.01	0.12	8333	2025
	chnl11.20	59	1824	0.30	>1000	2.78	0.16	0.01	0.30	9	>3226
XOR	x1_16	16	16	0.01	0.03	0.01	0.01	0.01	0.01	1	2
	x1_24	24	24	0.01	12.4	0.31	0.03	0.03	0.01	8	310
	x1_32	32	32	0.03	>1000	0.05	0.01	0.01	0.01	1	>25000
2pipe	2pipe_1.ooo	3	6	0.28	0.21	0.99	0.23	0.27	0.28	2	0
	2pipe_2.ooo	5	15	0.36	0.28	1.78	0.69	0.35	0.35	3	0
	2pipe	7	28	0.20	0.17	2.19	0.45	0.23	0.34	5	0
	Total	885	14802	3.77	12149	2088	8.05	3.22	8.35	339	5029
	Geometric Mean									24	344

Search runtimes are significantly smaller after augmenting the instances with symmetry-breaking predicates.

$supp(\pi)\backslash ends(\pi)$ and corresponds to a reduction in the number of BPs from n to $m \equiv |supp(\pi)| - cycles(\pi)$.

The third and last case corresponds to the BPs of those bits that occur *after* the first "phase-shifted variable." Let i be the index of the first variable for which $x_i^\pi = x_i'$. Thus, $e_i = 0$ and all BPs for $j > i$ have the form $0 \rightarrow (x_j \leq x_j^\pi)$, making them unconditionally true.

Taken together, the redundant BPs corresponding to these three cases can be easily eliminated by setting the index set in (12) and (13) to:

$$I = (supp(\pi)\backslash ends(\pi))\backslash succ(phase - shift(\pi), I_n). \qquad (17)$$

In the sequel, we will refer to the bits in the above index set as "irredundant bits." Note that the presence of a phase-shifted variable early in the total order can lead to a drastic reduction in the number of irredundant bits. For example, if $\pi = (x_1, x_1') \ldots$, then $PP(\pi)$ is simply (x_1'), regardless of how many other variables are moved by π.

TABLE 4
zChaff Search Runtimes of "Randomized" Hole-n Instances Augmented with Linear SBPs Based on Different Sets of Permutations

Instance	2-cycle generators		Generators with long cycles						
			Max cycle length	Generators only		Generators and their powers		Generators and their compositions	
	Total	Time (sec)		Total	Time (sec)	Total	Time (sec)	Total	Time (sec)
Hole-7	13	0.01	24	13	0.01	35	0.02	116	0.03
Hole-8	15	0.01	21	15	0.04	44	0.1	149	0.13
Hole-9	17	0.01	56	17	0.41	83	1.96	187	1.01
Hole-10	19	0.01	70	19	2.38	243	13.68	232	2.2

Total denotes the total number of permutations used in constructing each SBP.

4.2 Linear Construction of PPs through Chaining

The PP in (12) and (13) has a recursive structure that can be utilized to produce a CNF formula whose size is linear, rather than quadratic, in the cardinality of the index set I. Specifically, we introduce the "ordering" predicates $l_i = (x_i \leq x_i^\pi)$ and $g_i = (x_i \geq x_i^\pi)$ and, after algebraic manipulation, write the following equivalent expressions for the permutation predicate:

$$\begin{aligned} PP(\pi) &= g_0 \rightarrow \bigcap_{i \in I} \left\{ \left(\bigcap_{j \in pred(i,I)} g_j \right) \rightarrow l_i \right\} \\ &= g_0 \rightarrow l_k \wedge \left[g_k \rightarrow \bigcap_{i \in K} \left\{ \left(\bigcap_{j \in pred(i,K)} g_i \right) \rightarrow l_i \right\} \right], \end{aligned} \tag{18}$$

where $g_0 = 1$, $k = next(0, I)$, and $K = succ(k, I)$. Noting that, except for the index set used, the parenthesized expression on the second line of the above equation is identical to the expression on the first line, we introduce a sequence of *chaining predicates* $\{p_i | i \in I\}$ defined according to:

$$p_i = g_{prev(i,I)} \rightarrow \bigcap_{k \in K} \left[\bigcap_{j \in pred(k,K)} g_j \right] \rightarrow l_k, \tag{19}$$

where $K = \{i\} \cup succ(i, I) = \{k \in I | k \geq i\}$. The recursive structure of (18) now makes it possible to express each chaining predicate in terms of the one that follows it:

$$p_i = g_{prev(i,I)} \rightarrow l_i p_{next(i,I)} \quad i \in I, p_{n+1} \equiv 1 \tag{20}$$

and yields the following alternative representation of the permutation predicate:

$$PP(\pi) = p_{\min(I)} \wedge \bigcap_{i \in I} \left[p_i = g_{prev(i,I)} \rightarrow l_i p_{next(i,I)} \right], \tag{21}$$

which can be simplified further by replacing the equalities by one-way implications leading, finally, to:

$$PP(\pi) = p_{\min(I)} \wedge \bigcap_{i \in I} \left[p_i \rightarrow g_{prev(i,I)} \rightarrow l_i p_{next(i,I)} \right]. \tag{22}$$

The CNF representation of each conjunct in (22) is obtained by substituting the definitions of the l and g variables and using the distributive law. Thus, using this construction, the permutation predicate requires $|I|$ additional variables (the chaining predicates) and consists of $2|I|$ 3-literal and $2|I|$ 4-literal clauses for a total of $14|I|$ literals.

5 Experimental Results

We conducted a number of experiments to evaluate the effectiveness of the symmetry breaking constructions described above in reducing search times. We ran the experiments on representative CNF instances from the following six benchmark families:

1. **Hole-n:** Unsatisfiable pigeon-hole instances [7].
2. **Urq:** Unsatisfiable randomized instances based on expander graphs [21].
3. **GRoute:** Difficult satisfiable instances that model global wire routing in integrated circuits [2].
4. **FPGARoute and ChnlRoute:** Large satisfiable and unsatisfiable instances that model the routing of wires in the channels of field-programmable integrated circuits [14].
5. **XOR:** Various exclusive-or chains [16].
6. **2pipe:** Difficult unsatisfiable instances that model the functional correctness requirements of modern out-of-order microprocessor CPUs [22].

Each of the benchmarks was converted to a colored graph, as described in Section 3, and processed by the graph automorphism program Saucy [6]. The symmetries returned by Saucy were then mapped back to symmetries of the benchmark and appropriate SBPs constructed and added. The zChaff SAT solver [13] was then run on the original and SBP-augmented versions of each benchmark. All experiments were run on a Linux workstation with a 2Ghz Pentium 4 processor and 1GB of RAM. A time-out limit of 1,000 seconds was set for all runs.

Table 1 lists, for each benchmark family, the name of the tested instance (column 2), its total CNF size (columns 3, 4, and 5), the order of its symmetry group (column 6), the total number of generators returned by Saucy (column 7), and the number of those that include phase shifts (column 8). Columns 9 and 10 list the cardinality of the generators' support and the degree of sparsity present in these generators. Table 2 lists the CNF sizes of three SBP constructions based on generators:

TABLE 5
Comparison of Search Runtimes for Various SAT Solvers with and without SBPs

34	35	36	37	38	39	40	41	42
Bench-mark Family	Instance	Time to find symmetries (sec)	BerkMin			miniSAT		
			Time to solve original instance (sec)	Time to solve instances and SBPs (sec)	$\frac{\text{col 37}}{\text{col 36 + col 38}}$	Time to solve original instance (sec)	Time to solve instances and SBPs (sec)	$\frac{\text{col 40}}{\text{col 36 + col 41}}$
Hole-n	hole07	0.00	0.07	0.00	700	0.07	0.00	25
	hole08	0.00	0.42	0.00	4200	0.57	0.00	286
	hole09	0.01	2.67	0.00	267	6.31	0.00	420
	hole10	0.02	45.29	0.00	2265	52.34	0.00	2181
	hole11	0.02	>1000	0.00	>50000	>1000	0.01	>38463
	hole12	0.02	>1000	0.00	>50000	>1000	0.01	>38463
Urq	Urq3_1	0.02	89.19	0.04	1487	6.13	0.02	143
	Urq3_4	0.01	0.10	0.00	10	0.33	0.00	23
	Urq3_5	0.04	>1000	0.22	>3846	119.09	0.34	317
	Urq3_9	0.01	0.79	0.00	79	0.97	0.00	69
GRoute	grout3.3-03	0.51	7.84	0.24	10	0.04	0.14	0
	grout3.3-04	0.46	4.25	0.11	7	7.00	0.35	9
	grout3.3-10	0.55	20.98	0.47	21	7.53	0.64	6
FPGARoute	fpga10.08	0.02	0.03	0.00	2	0.00	0.00	0
	fpga10.09	0.03	0.01	0.00	0	0.00	0.01	0
	fpga12.08	0.03	0.02	0.00	1	0.00	0.01	0
	fpga12.09	0.04	0.03	0.00	1	0.00	0.01	0
	fpga12.11	0.07	0.04	0.01	1	0.00	0.01	0
	fpga12.12	0.08	0.12	0.00	2	0.00	0.01	0
	fpga13.09	0.05	0.06	0.00	1	0.01	0.01	0
	fpga13.10	0.06	0.02	0.00	0	0.00	0.01	0
	fpga13.12	0.10	0.05	0.00	1	0.00	0.01	0
ChnlRoute	chnl10.11	0.06	57.95	0.00	966	76.01	0.01	1102
	chnl10.12	0.08	>1000	0.00	>12500	48.36	0.01	543
	chnl10.13	0.09	>1000	0.00	>11111	35.41	0.01	365
	chnl11.12	0.09	>1000	0.00	>11111	>1000	0.01	>10204
	chnl11.13	0.11	>1000	0.00	>9091	>1000	0.01	>8265
	chnl11.20	0.30	>1000	0.00	>3333	>1000	0.02	>3165
XOR	x1_16	0.01	0.01	0.00	1	0.04	0.00	3
	x1_24	0.01	2.92	0.04	58	1.81	0.02	58
	x1_32	0.03	10.66	0.00	355	5.15	0.01	143
2pipe	2pipe_1.ooo	0.28	0.06	0.07	0	0.09	0.10	0
	2pipe_2.ooo	0.36	0.07	0.09	0	0.09	0.17	0
	2pipe	0.20	0.05	0.08	0	0.33	0.13	1
	Total	3.77	8244	1.37	161K	5368	2.09	104K
	Geometric Mean				50			16

Search runtimes are significantly smaller after augmenting the instances with symmetry-breaking predicates.

- The quadratic construction (using extra equality variables) based on all bits; this represents the previous state-of-the-art.
- The quadratic construction based only on irredundant bits.
- The linear construction (using extra chaining variables) based only on irredundant bits.

Several observations can be made about the data in Tables 1 and 2. The number of symmetries in these benchmarks is large, but all symmetries, including phase shifts in benchmark families *Urq, XOR,* and *2pipe*, can be represented by fairly small sets of generators. The generators returned by Saucy appear very sparse on average, i.e., a typical generator affects only a small number of variables. This explains the reduction, by 1-2 orders-of-magnitude, in the size of symmetry-breaking predicates in column 18 (our first construction) versus column 15 ([5]): The number of variables, clauses, and literals is reduced. While our construction in column 18 only slightly extends the quadratic-size construction in [1], our more advanced linear-size construction (column 21) offers an additional reduction by up to an order of magnitude. Note, however, that the number of variables is unchanged—the extra variables added by the two constructions have different function, but can be mapped to each other one-to-one.

Table 3 empirically compares the effectiveness of the symmetry-breaking predicates described in Table 1. First, in most cases, it takes much less time to find symmetries of a CNF instance than to solve it. The *2pipe* instances are an exception, but we believe that further advances in symmetry-finding can rectify this exception. Second, the all-bits quadratic-sized construction due to [5] is dramatically slower than our variants, based on the cycle structure. Our linear-sized construction provides a further speed-up. The only exception is the *2pipe_1_ooo* instance, where the difference between the irredundant-bits linear and quadratic-sized constructions is small.

Table 3 offers additional data to evaluate symmetry-breaking by generators, which may not be complete. We added symmetry-breaking predicates built for pairwise products of generators, but the overall runtimes increased in most cases. While additional SBPs may break more symmetries, their overhead does not justify their use.

Table 4 describes experiments with generators that have long cycles in which we evaluated extensions to symmetry-breaking by generators. Namely, we tried adding powers of all generators and, alternatively, adding pairwise products of generators. Neither extension proved useful, which supports our main symmetry-breaking approach.

In order to study the effect of symmetry breaking when using other state-of-the-art SAT solvers, we solved the instances using two of the best known SAT solvers: BerkMin562 [10] and miniSAT v1.14 [8]. The default settings were used with both solvers. We used the advanced linear-size construction for generating the SBPs. Table 5 shows BerkMin's and miniSAT's runtimes when solving the instances with and without symmetry-breaking predicates. For both solvers, the addition of SBPs leads to significant runtime savings (8,243 versus 1.37 seconds for BerkMin and 5,367 versus 2.09 for miniSAT).

We noticed that running local search solvers, e.g., WalkSAT [17], with symmetry-breaking clauses does not improve runtimes. In some cases, it makes runtimes worse, which was also observed by Prestwich in [15].

In terms of complexity, the processing of CNF-SAT instances which results in the addition of SBPs includes symmetry-finding, for which no polynomial-time algorithms are currently known in the general case (but the graph automorphism problem solved as a step is not believed to be NP-complete unless P = NP). However, symmetry-finding is often performed very quickly in practice. Given symmetry-generators, we build one SBP per symmetry-generator. One can show that, for a graph with N vertices, the maximal number of symmetry-generators returned is $N^2(\log_2 N)^2$, therefore, for a CNF instance with V variables and C clauses, the number of irredundant generators is $O((V+C)^2(\log_2(V+C))^2)$. In practice, symmetries are often represented more compactly and the number of generators is much smaller than V. For example, all permutational symmetries of k variables can be captured by just two generators.

To estimate the size of an SBP, we build for a given symmetry generator, we only use its action on vertices and ignore the permutation of clauses that it performs. Since vertices that are mapped onto themselves do not affect the size of an SBP, the size is a function of the vertex-based support of the symmetry-generator. While this support may include all vertices, in practice, it is typically much smaller, e.g., many generators are transpositions ($|supp| = 2$) or small sets of transpositions. When we build a new SBP, the number of literals in added clauses grows linearly with the size of the vertex-based support of the symmetry generator —this is in contrast to quadratically-growing SBPs in the previous literature. Our algorithm to produce SBPs also has linear asymptotics.

6 Conclusions

The main contribution of our work is a better construction of symmetry-breaking predicates for Boolean satisfiability. We empirically demonstrate improvements both in the size of predicates and the runtime of SAT solvers after these predicates are added to the original CNF instances. We also show that 1) symmetry-breaking by generators is difficult to improve upon and that 2) the efficiency of symmetry-breaking does not improve when larger cycles are found in generators.

Our work articulates that better symmetry finding algorithms would be useful, especially if tailored to CNF formulas and, perhaps, the kinds of symmetry groups commonly found in structured CNF instances.

Acknowledgments

This work was funded by the DARPA/MARCO Gigascale Silicon Research Center and an Agere Systems/SRC Research fellowship.

References

[1] F. Aloul, A. Ramani, I.L. Markov, and K. Sakallah, "Solving Difficult Instances of Boolean Satisfiability in the Presence of Symmetries," *IEEE Trans. Computer Aided Design,* vol. 22, no. 9, pp. 1117-1137, 2003.

[2] F. Aloul, A. Ramani, I.L. Markov, and K. Sakallah, "Generic ILP versus Specialized 0-1 ILP," *Proc. Int'l Conf. Computer-Aided Design (ICCAD),* pp. 450-457, 2002.

[3] R. Bayardo Jr. and R. Schrag, "Using CSP Look-Back Techniques to Solve Real World SAT Instances," *Proc. Nat'l Conf. Artificial Intelligence,* pp. 203-208, 1997.

[4] A. Biere, A. Cimatti, E. Clarke, M. Fujita, and Y. Zhu, "Symbolic Model Checking Using SAT Procedures instead of BDDs," *Proc. Design Automation Conf. (DAC),* pp. 317-320, 1999.

[5] J. Crawford, M. Ginsberg, E. Luks, and A. Roy, "Symmetry-Breaking Predicates for Search Problems," *Proc. Int'l Conf. Principles of Knowledge Representation and Reasoning,* pp. 148-159, 1996.

[6] P. Darga, M.H. Liffiton, K.A. Sakallah, and I.L. Markov, "Exploiting Structure in Symmetry Generation for CNF," *Proc. Design Automation Conf. (DAC),* pp. 530-534, 2004.

[7] DIMACS Challenge benchmarks, ftp://Dimacs.rutgers.EDU/pub/challenge/sat/benchmarks/cnf, 1996.

[8] N. Een and N. Sorensson, "An Extensible SAT-Solver," *Theory and Applications of Satisfiability Testing,* pp. 502-518, 2003.

[9] J.B. Fraleigh, *A First Course in Abstract Algebra,* sixth ed. Reading, Mass.: Addison Wesley Longman, 2000.

[10] E. Goldberg and Y. Novikov, "BerkMin: A Fast and Robust SAT-Solver," *Proc. Design, Automation, and Test in Europe Conf. (DATE),* pp. 142-149, 2002.

[11] T. Larrabee, "Test Pattern Generation Using Boolean Satisfiability," *IEEE Trans. Computer Aided Design,* vol. 11, no. 1, pp. 4-15, 1992.

[12] B. McKay, "Practical Graph Isomorphism," *Congressus Numerantium,* vol. 30, pp. 45-87, 1981, http://cs.anu.edu.au/bdm/nauty/.

[13] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an Efficient SAT Solver," *Proc. Design Automation Conf. (DAC),* pp. 530-535, 2001.

[14] G. Nam, F. Aloul, K. Sakallah, and R. Rutenbar, "A Comparative Study of Two Boolean Formulations of FPGA Detailed Routing Constraints," *Proc. Int'l Symp. Physical Design (ISPD),* pp. 222-227, 2001.

[15] S. Prestwich, "Supersymmetric Modeling for Local Search," *Proc. Workshop Symmetry and CSPs (SymCon),* 2002.

[16] SAT 2002 Competition, http://www.satlive.org/SATCompetition/submittedbenchs.html, 2002.

[17] B. Selman, H. Kautz, and B. Cohen, "Noise Strategies for Improving Local Search," *Proc. Nat'l Conf. Artificial Intelligence,* pp. 337-343, 1994.

[18] J. Silva and K. Sakallah, "GRASP: A New Search Algorithm for Satisfiability," *IEEE Trans. Computers,* vol. 48, no. 5, pp. 506-521, May 1999.

[19] E. Spitznagel, "Review of Mathematical Software, GAP," *Notices Am. Math. Soc.,* vol. 41, no. 7, pp. 780-782, 1994.

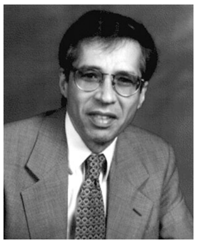
[20] P. Stephan, R. Brayton, and A. Sangiovanni-Vincentelli, "Combinational Test Generation Using Satisfiability," *IEEE Trans. Computer-Aided Design,* vol. 15, no. 9, pp. 1167-1175, 1996.

[21] A. Urquhart, "Hard Examples for Resolution," *J. ACM,* vol. 34, no. 1, pp. 209-219, 1987.

[22] M.N. Velev and R.E. Bryant, "Effective Use of Boolean Satisfiability Procedures in the Formal Verification of Superscalar and VLIW Microprocessors," *Proc. Design Automation Conf. (DAC),* pp. 226-231, 2001.

[23] H. Zhang, "SATO: An Efficient Propositional Prover," *Proc. Int'l Conf. Automated Deduction,* pp. 272-275, 1997.

Fadi A. Aloul (S'97-M'03) received the BS degree in electrical engineering (summa cum laude) from Lawrence Technological University, Southfield, Michigan, in 1997 and the MS and PhD degrees in computer science and engineering from the University of Michigan, Ann Arbor, in 1999 and 2003, respectively. He was a postdoctoral research fellow at the University of Michigan during the summer of 2003. He is currently an assistant professor of computer engineering at the American University of Sharjah (AUS), Sharjah, United Arab Emirates. In the summer of 2005, he was a visiting researcher with the Advanced Technology Group at Synopsys, Portland, Oregon. He has published more than 40 papers and has presented invited talks at several industrial sites. He has also developed several tools for Boolean satisfiability, including the pseudo-Boolean SAT solver and optimizer PBS. His research interests are in the areas of computer-aided design, verification, combinatorial optimization, and Boolean satisfiability. He has received a number of awards, including the Agere/SRC research fellowship, GANN fellowship, and the LTU presidential scholarship. He served on the technical program committees at the International Workshop on Logic Synthesis (IWLS), the International Conference on Theory and Applications of Satisfiability Testing (SAT), the SIGDA PhD Forum at the Design Automation Conference, the International Workshop on Soft Constraints (CP-Soft), and the International Symposium on Wireless Systems and Networks. He was the AV chair of the 2003 International Workshop on Logic Synthesis (IWLS). He is a member of the IEEE, the ACM, and Tau Beta Pi.



Karem A. Sakallah (S'76-M'81-SM'92-F'98) received the BE degree in electrical engineering from the American University of Beirut, Beirut, Lebanon, in 1975, and the MSEE and PhD degrees in electrical and computer engineering from Carnegie Mellon University, Pittsburgh, Pennsylvania, in 1977 and 1981, respectively. In 1981, he was with the Department of Electrical Engineering at Carnegie Mellon University as a visiting assistant professor. From 1982 to 1988, he was with the Semiconductor Engineering Computer-Aided Design Group at Digital Equipment Corporation in Hudson, Massachusetts, where he headed the Analysis and Simulation Advanced Development Team. Since September 1988, he has been with the University of Michigan, Ann Arbor, as a professor of electrical engineering and computer science. From September 1994 to March 1995, he was with the Cadence Berkeley Laboratory in Berkeley, California, on a six-month sabbatical leave. He has authored or coauthored more than 200 papers and has presented seminars and tutorials at many professional meetings and various industrial sites. His current research interests include the area of computer-aided design with emphasis on logic and layout synthesis, Boolean satisfiability, discrete optimization, and hardware and software 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 numerous other workshops. He is currently an associate editor of the *IEEE Transactions on Computers*. He is a fellow of the IEEE and a member of the ACM and Sigma Xi.

Igor L. Markov (M'97-SM'05) received the MA degree in mathematics (1994) and the PhD degree in computer science (2001), both from the University of California Los Angeles. He is currently an assistant professor in the Department of Electrical Engineering and Computer Science at the University of Michigan, Ann Arbor. He published more than 100 refereed papers in journals, magazines, and conference proceedings. His interests include combinatorial optimization with applications to the design and verification of integrated circuits, as well as quantum logic circuits. He is a senior member of the IEEE and a member of the ACM and the AMS. In 2001, he was awarded the DAC Fellowship and received the IBM University Partership Award. He received the 2004 IEEE CAS Donald O. Pederson paper-of-the-year award and the 2004 ACM SIGDA Outstanding New Faculty Award. He won the best paper award at DATE 2005 in the Circuit Test category, the US National Science Foundation Career Award, the Synplicity Inc. Faculty Award, and the 2005 SCM SIGDA Technical Leadership Award. He served on the technical program committees at the Design Automation Conference, International Conference on Computer-Aided Design, Design Automation and Test in Europe Conference, International Symposium on Physical Design, and several other IEEE conferences and symposia. He served as the general chair and the technical program committee chair of the International Workshop on System-Level Interconnect Prediction. Currently, he is serving as a guest editor of *Integration: The VLSI Journal*