SBSAT: A State-Based, BDD-Based Satisfiability Solver

John Franco[1], Michal Kouril[1], John Schlipf[1], Jeffrey Ward[1], Sean Weaver[1], Michael Dransfield[2], and W. Mark. Vanfleet[2]

[1] The University of Cincinnati, Cincinnati 45221-0030, USA John.Franco@UC.Edu
[2] Department of Defense, United States of America

Abstract. We present a new approach to SAT solvers, supporting efficient implementation of highly sophisticated search heuristics over a range of propositional inputs, including CNF formulas, but particularly BDDs. The approach memoizes search information using a new form of lookahead, called *local-function-complete* lookahead, during an extensive preprocessing phase; we expect to do further experimentation on heuristics made feasible by the memoization. Preprocessing also includes manipulating inputs to more exploitable form. This approach has been incorporated, along with existing tools such as lemmas, to build a SAT tool we call SBSAT. We show the feasibility of SBSAT by comparing it to zChaff on several of the benchmarks. We also show an interesting dependence of some standard benchmarks upon simply the independent/dependent variable distinction.
Keywords: Satisfiability, State Machine, Binary Decision Diagram, DAG.

1 Introduction

Recent development of technologies for solving the propositional Satisfiability (PSAT) problem has been so successful it has captured the attention of people working in areas as diverse as theoretical physics and computer engineering. Significant among those technologies are the use of conflict-resolution or lemmas to turn a tree search into a DAG search, the development of advanced "lemma heuristics" for choosing the "best" lemmas, partial lookahead for information that can be used to enhance "search heuristics", non-chronological backtracking, and advanced data structures [LMS02,ZMMM01]. Newer technologies, *e.g.* based on autarkies [Kul98] and symmetry [Gol02,GN02], show great promise.

But many PSAT problems still are difficult, and many of those do not naturally appear as CNF problems. One can translate them to CNF and apply a CNF solver. This translation need not expand the formula by more than a constant factor [Sch89], but new variables must be added to achieve this. Moreover, some information may be hidden by the translation, such as clustering of dependencies or distinctions between so-called independent and dependent variables. Exploiting this information may speed up a search.

Simply to emphasize that we are allowing non-CNF input, we shall refer to our problems as *PSAT* rather than just SAT. One standard representation for complex boolean functions is the *Reduced Ordered Binary Decision Diagram* (*BDD*) [Bry86]. A BDD is a canonical DAG representation of a boolean function in terms of constants 0 and 1 and

the if-then-else operator *ite*. For example, the tree representation of $ite(x_1, ite(x_2, 1, 0), 0)$, with the two 0 nodes are merged, is the BDD expressing $x_1 \wedge x_2$. Many problems in microprocessor design, for example, such as in design verification and interconnect synthesis, are more naturally expressed with BDD constraints rather than CNF constraints. Standard logical operations are easily implemented on BDDs, and BDDs have been used successfully in many cases over the past 10 years. However, as the number of variables grow, BDDs can grow exponentially, limiting pure BDD methods.

An obvious next step is to develop a hybrid algorithm, combining BDD tools and Davis-Putnam-Loveland-Logemann (*DPLL*) search [DLL62].[3] We propose here a new variety of hybrid. Typically, the input will be a PSAT expressed in terms of BDD's. We also take full advantage of the huge memory capacity now routinely available on general purpose, low-cost computers, to precompute (compute once, before starting the brancher) as much as feasible.[4]

1. Do as much BDD-type preprocessing as is feasible. We define two new BDD operations for simplifying a collection of BDDs while avoiding size explosion: *strengthening* and *branch pruning* (Sect. 2).
2. Before applying a DPLL-style search procedure, precompute as much static information as possible, to speed backtracking. (Sect. 3.)
3. Use a new search heuristic for choosing branching variables: A single BDD can encode complex relationships among its variables; precompute *complete* lookahead information for all its partial truth assignments, and then combine that lookahead information accross input BDDs at branch time (Sect. 3). We call this *local-function-complete* lookahead

2 BDD Preprocessing

SBSAT first preprocesses the input formulas, typically BDDs, before preparing for the DPLL-type search. We borrow and modify techniques from BDD solvers, avoiding techniques that will explode BDD size.

Individual BDDs not broken into CNF formulas may force some variables to be true or false or force some literals to be equivalent. We identify this and simplify the input. If a variable appears in only one BDD — in only b_1 among $b_1, \ldots, b_m$ below — we may use Boolean *existential quantification*: $\exists x_i(b_1 \wedge \cdots \wedge b_k)$ is logically equivalent to $\exists x_i(b_1) \wedge b_2 \wedge \cdots \wedge b_m$. We can now solve the 1-fewer variable problem and choose x_i to satisfy b_1 after the search is done.

A simple BDD solver, given BDDs $b_1, b_2, \ldots, b_k$, may conjoin them, resulting in BDD size explosion. We *strengthen* each b_i: conjoin it with the projections of all other constraints onto its variables; this may letting us infer literals or equivalences early.

At other times it is useful to decouple conjunctions. Given two BDD's b_1, b_2, we *branch prune* duplicated logic, removing from BDD b_2 all branches that contradict b_1.[5]

[3] For other approaches, see, *e.g.* [PG96,GA98,PK00,KZCH00].

[4] Preprocessing in SAT solvers is not new, but our automata (see Sect. 3) provide new ways to use memoization.

[5] The algorithm is a modification of Brace's *generalized cofactor* algorithm on BDD's [Bra90].

There appear to be two gains: It can make our state machines (see Sect. 3) smaller. And it often *appears*, by avoiding logic, to make our local-function-complete lookahead heuristic's evidence combination rule work better. (However, it can, in odd cases, also lose local information.)

We provide user control for how much preprocessing to do, presuming the user can learn what works well on what classes of problems.

3 State Machines and the LSGB Heuristic

We normally preprocess boolean constraints into acyclic Mealy machines called *SMURF*s (for "State Machine Used to Represent Functions"). We may assume each constraint implies no literals, since those would have been trapped during preprocessing. *SMURF*s are described in Fig. 3. For a set of constraint BDDs, we compute the *SMURF*s for each of the separate BDDs and merge states with equal residual functions, maintaining one pointer into the resultant automaton for the current state of each constraint. For each single boolean function there is a unique such state machine.

We precompute information for choosing branching variables. The weight of a transition is the number of literals forced on the transition plus the expected number of literals forced below that state, where a forced literal after m additional choices is weighted $1/K^m$. (K, set experimentally, is currently 3.) In Fig. 3, the transition out of the start state on $\neg x_1$ has weight $1 + (\frac{1}{K} + \frac{1}{K} + \frac{1}{K} + \frac{1}{K})/4$; the transition out on x_4, $0 + (\frac{1}{K}^2 + \frac{2}{K} + \frac{1}{K} + \frac{2}{K} + \frac{2}{K} + \frac{1}{K})/6$. At brancher time we need only look up these individual weights in a table.

Our "locally-skewed, globally-balanced" (*LSGB*) search heuristic, for $K = 2$, is similar to the "Johnson heuristic" in CNF. Branch toward forced inferences as quickly as possible, narrowing the search space and getting lemmas fast. For each variable x_i, compute (i) the sum S_i^+ of the *weight*s of transitions on x_i out of all current *SMURF* states, and (ii) the sum S_i^- of the *weight*s of transitions on $\neg x_i$; a high sum represents a high "payoff." Ideally for branching, both x_i and $\neg x_i$ force many literals; we branch on the variable x_i maximizing $S_i^+ \cdot S_i^-$; branch first toward the larger of S_i^+, S_i^-.[6]

LSGB is intended for applications where little is known about — or easily determined about — the given PSAT problem. It performs well there. If a problem is known to have a lot of exploitable structure, it may be better to specify a different heuristic; we allow experienced users some choice. *SMURF*s admit many heuristics; on a simple heuristic, at worst, (except for preprocessing time) they do not hinder. Work is needed on hybrid heuristics.

4 Lemmas

Except for data structures and search heuristics, SBSAT generalizes standard DPLL-type searches. Having *SMURF*s output forced literals allows generalizing unit clause propagation. SBSAT also makes extensive use of backjumping, recent advanced data structures, and lemmas.

[6] We borrow the idea of taking the product from Freeman [Fre95].

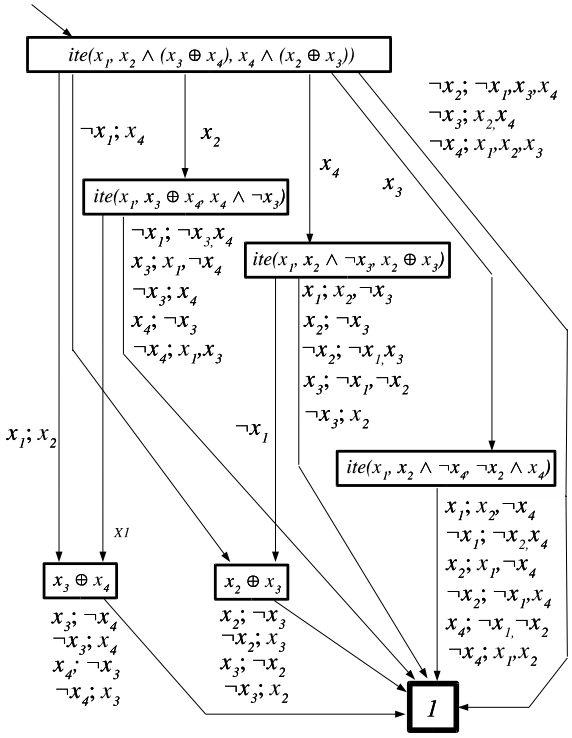

Fig. 1. We preprocess BDDs into deterministic Mealy machines called "*SMURF*s." This example explains construction. *ite* denotes if-the-else and $\oplus$ denotes exclusive or.

The *SMURF* represents $ite(x_1, x_2 \wedge (x_3 \oplus x_4), x_4 \wedge (x_2 \oplus x_3))$. It represents, in part, BDDs for the function under all possible variable orderings — since we cannot know in what order the brancher considers the variables.

The start state (upper left) represents the original function. On the left is a transition from the start state labeled "$x_1; x_2$"; this means that, from that state, on input x_1, the automaton makes a transition and outputs $\{x_2\}$. If the brancher guesses, or infers, that x_1 is true, it will "tell" the automaton to branch on x_1. The output of x_2 tells the brancher that x_2 must also be true — the analogue of unit inference in CNF. This transition goes to a state labeled $x_3 \oplus x_4$, meaning that, after x_1, x_2 are set to 1, what remains to be satisfied — the *residual function* — is $x_3 \oplus x_4$. On the upper right are three transitions shown with one arrow. The first is from the start state on input $\neg x_2$; it outputs $\{\neg x_1, x_3, x_4\}$ and goes to state 1 — meaning the original BDD is now satisfied, *i.e.*, that there is no residual constraint to satisfy.

Some user problems, *e.g.*, the dlx benchmark suite made available by Miroslaw Velev [Vel00], use very long clauses and long assignments $\lambda_0 = \lambda_1 \wedge \cdots \wedge \lambda_k$ or $\lambda_0 = \lambda_1 \vee \cdots \vee \lambda_k$ (where the λ_is are literals). To save space while keeping a formula in a single structure (to maximize functional-complete-lookahead), we have separate data structures for storing these three forms (also long linear expressions) using counters, plus lookup tables to for the transition weights as functions of those counters (exactly as if in the Mealy machines).

Otherwise, to avoid extreme state explosion — at worst, the SMURF for an n variable boolean function may have almost 3^n states — we limit individual constraints to 17 variables. (We expect to add routines to split functions of more variables automatically by introducing defined variables). This limitation has been of little significance so far; indeed, we believe that on some practical benchmarks SBSAT suffers from the problems being broken down too far for it to take advantage of input data clustering.

SBSAT creates clause lemmas, not BDD lemmas, for efficiency. It creates lemmas lazily — during branching — memoizing them in a lemma cache.[7] SBSAT creates a lemma when a literal is forced and resolves lemmas during backtracking. Lemmas that seem useful are cached. During the search, if a partial assignment negates all but one literal of a clause, that last literal is inferred true. SBSAT uses a modified Chaff-type data structures for the cache [LMS02]. Chaff restarts when it fills its lemma cache; SBSAT continues, deleting the lemma least recently used. More work is needed here.

5 Computational Results

Our primary interest has been to provide a platform for sophisticated search heuristics. We report work on two sets of benchmark problems: a set of random "sliding window" problems, and the dlx benchmark suite made available by Miroslaw Velev [Vel00]). We compared SBSAT to zChaff [MMZZM01], the most successful CNF solver to date on several of the dlx problem sets.

The dlx problem sets, arising from microprocessor-verification work at Carnegie Mellon, seemed almost prohibitively difficult before Chaff. Now Chaff versions can solve them all relatively easily. We show below that, with suitable tuning, SBSAT can be competitive with zChaff.

"Suitable tuning" involves two steps: The LSGB heuristic, tailored for a different sort of problem, was replaced with a simplification "nChaff" (near Chaff) of Chaff's heuristic. We see below that, on these problem sets, much of what zChaff finally exploits is the difference between independent and dependent variables. Here a *dependent variable* is one which the user *defines*, in the trace format versions of the dlx set, in terms of an assignment. On dlx, having SBSAT always branch on dependent variables before independent variables can speed up the search massively.[8] There are other problems where branching on independent variables first significantly speeds up the search. This suggests the possibility of dovetailing choices, alternating between branching on independent and dependent variables. Importantly, by staying in the user domain rather than CNF, SBSAT can easily separate variables which the user describes as dependent.

Times reported below include preprocessing times. SBSAT input was in CMU's trace format, not CNF, allowing for automatic detection of dependent variables. Measurements were taken on a 2GHz Pentium 4/Linux v. 2.4.7 platform with 2GB RAM. SBSAT's lemma cache size was set to 20000.

Now zChaff still is faster; *e.g.,* on `dlx2_cc` it runs over four times as fast as the best SBSAT run. This is due to zChaff's simplicity — and thus fast execution: it makes 24,305 decisions, whereas, with dependent-first variable choice, SBSAT makes 25,921

[7] We have not yet incorporated Chaff's "critical path analysis."

[8] This is in conflict with a frequent intuition that we should branch first on independent variables, since they are forced anyway. There has indeed been a fair amount of discussion about independent variables and dependent variables and whether to branch on one type before the other [KMS97,CGPRST01,Sht00,GS99]. We do not know whether this effect is intrinsic to the logic of these circuit design problems or to the way the designer thought about them. Of course, we also do not know how zChaff would perform if the user were allowed to input domain knowledge — in this case that it is wise to branch on dependent variables first.

Table 1. Times (in seconds) for zChaff and SBSAT on some dlx problems.

Benchmark	zChaff	SBSAT /LSGB	SBSAT /LSGB depndt-1st	SBSAT nChaff	SBSAT nChaff depndt-1st
Satisfiable:					
dlx2_cc_bug01	1.00	347.92	12.31	6.22	5.25
dlx2_cc_bug08	1.13	3.12	2.99	3.64	3.01
dlx2_cc_bug40	0.94	272.38	10.01	14.78	4.34
Unsatisfiable:					
dlx2_dlx2_aa	0.14	1.98	0.99	2.34	0.92
dlx2_dlx2_cc	1.43	1361.02	18.39	14.76	6.18

backtracks under LSGB and 15,079 backtracks under nChaff. But SBSAT is becoming competitive.

Recall also that SBSAT gives the user a choice of heuristics, letting the user gain experience with what kinds of heuristic help on various classes of problems. We believe that the success of the dependent-variables-first heuristic illustrates the utility of providing that choice.

As noted earlier, regardless of whether the LSGB or nChaff search heuristic is employed, great performance for SBSAT seems to depend more on choosing dependent variables first than on the search heuristic. From the way the zChaff results follow the SBSAT results and considering that SBSAT does not take into account any weighting of lemmas as zChaff does when choosing a variable for branching, one might suppose that the strength of zChaff on the dlx benchmarks is more due to zChaff's heuristic happening to choose dependent variables before independent variables than to restarts, data structures, or any other feature of zChaff. We leave investigation of this remark to a future paper.

Our goal was partly to build a solver for PSAT problems not handled well by other solvers — not dealt with by traditional CNF tools, such as lemmas. The "sliding window" problems (below) were devised to be computationally hard for these solvers. In its current form, zChaff is somewhat handicapped by its clever data structures for lemma handling, which limit the search heuristics it can use effectively. The strength of the heuristic power SBSAT (using LSGB), relative to zChaff, is demonstrated by these "sliding window" problems.

Generate m constraints over m variables (m even) as follows: Pick random boolean functions, $f(v_1, v_{i_1}, \ldots, v_{i_k}, v_{m/2})$, $g(v_1, v_{j_1}, \ldots, v_{j_l}, v_{m/2})$ (with variables explicitly listed, in order of subscript, and k, l are relatively small). The constraint set is
$\{f(v_{1+h}, f_(v_{i_2+h} \ldots, v_{i_k+h}, v_{\frac{m}{2}+h}) : 0 \leq h \leq (m/2)\} \cup$
$\{g(v_{1+h}, v_{j_2+h}, \ldots, v_{m/2+h}) = o_h : 0 \leq h \leq (m/2)\}$, where each o_h is randomly chosen to be 0 or 1. This provides distinct pattern to the data but gives no global impor-

tance to one single variable over another; fast search depends upon identifying variable groups, not single variables.[9] Experimental results are shown in Table 5.

Table 2. Times for zChaff and SBSAT on sliding windows problems. SBSAT lemmas were disabled.

#Variables	Satisfiable?	zChaff (seconds)	SBSAT (seconds)
60	sat	0.15	0.76
60	unsat	1.74	1.05
80	sat	1.00	1.38
80	unsat	149.53	9.98
100	sat	8.92	1.47
100	unsat	2288.11	153.70
120	sat	> 10000	89.90
120	unsat	> 10000	4259.74

On the small examples, SBSAT is slower than zChaff; this is due entirely to preprocessing time. Thus examples *seem* to illustrate that, when there are few key variables in a problem for zChaff to discover as it builds lemmas, zChaff's search is highly inefficient. This motivates using another paradigm, such as an heuristic based upon local-function-complete lookahead, in such circumstances.

Our goal in this project was to provide a suite of tools to approach problems beyond the scope of other current solvers. To this end we provide the user with choice of option, so that the user may exploit domain knowledge. We have also tried to deal with problems, such as the "sliding window" problems above, which are particularly difficult for current solvers.

This research was partially supported by U.S. Department of Defense grant MDA 904-02-C-1162.

References

[Bra90] Brace, K.S., R.L. Rudell, and R.E. Bryant.: Efficient Implementation of a BDD Package. *ACM Proceedings of the 27th ACM/IEEE Design Automation Conference* (1990) 40–45.

[Bry86] Bryant, R.E.: Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers* **C-35** (1986) 677–691.

[CGPRST01] Cimatti, A., E. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella.: NuSMV Version 2: BDD-Based + SAT-Based Symbolic Model Checking. Available from `http://sra.itc.it/people/roveri/papers/IJCAR01.ps.gz`.

[DLL62] Davis, M., G. Logemann, and D. Loveland.: A Machine Program for Theorem Proving. *Communications of the Association of Computing Machinery* **5** (1962) 394–397.

[9] By contrast, as noted above, we conjecture that Chaff is effective largely because it identifies key variables, such as dependent variables.

[Fre95] Freeman, J.: Improvements to Propositional Satisfiability Search Algorithms. Ph.D. Dissertation in Computer and Information Science, University of Pennsylvania, 1995.

[GS99] Giunchiglia, E., and R. Sebastiani.: Applying the Davis-Putnam Procedure to Non-Clausal Formulas. *Proceedings of AI*IA '99, Lecture Notes in Artificial Intelligence, #1792*, Springer Verlag, 1999.

[Gol02] Goldberg, E.: Testing Satisfiability of CNF Formulas by Computing a Stable Set of Points. *Proceedings of the Fifth International Symposium on the Theory and Applications of Satisfiability Testing* (2002) 54–69. (Available from: `http://gauss.ececs.uc.edu/Conferences/SAT2002/Abstracts/goldberg.ps` Expanded version submitted for publication.)

[GN02] Goldberg, E., and Y. Novikov.: BerkMin: A Fast and Robust Sat-Solver Design. *Proceedings Design, Automation, and Test in Europe* 2002 142–149.

[GA98] Gupta, A., and P. Ashar.: Integrating a Boolean Satisfiability Checker and BDDs for Combinational Equivalence Checking. *Proceedings 11th IEEE International Conference on VLSI Design: VLSI for Signal Processing* (1998) 222–225.

[KZCH00] Kalla, P., Z. Zeng, M.J. Ciesielski, and C. Huang.: A BDD-based Satisfiability Infrastructure Using the Unate Recursive Paradigm. *Proceedings Design, Automation, and Test in Europe* 2000 232–236.

[KMS97] Kautz, H., D. Mc Allester, and B. Selman.: Exploiting Variable Dependency in Local Search. Available from `http://www.cs.washington.edu/homes/kautz/papers/dagsat.ps`.

[Kul98] Kullmann, O. Heuristics for SAT Algorithms: Searching for Some Foundations. Submitted for publication. (Available from `http://cs-svr1.swan.ac.uk/~csoliver/heur2letter.ps.gz`).

[LMS02] Lynce, I., and J. Marques-Silva.: Efficient Data Structures for Backtrack Search SAT Solvers. *Proceedings of the Fifth International Symposium on the Theory and Applications of Satisfiability Testing* (2002) 308–315. (Available from: `http://gauss.ececs.uc.edu/Conferences/SAT2002/Abstracts/lynce.ps`.

[MMZZM01] Moskewicz, M.W., C. Madigan, Y. Zhao, L. Zhang, and S. Malik.: Engineering a (Super?) Efficient SAT Solver. *Proceedings of the 38th ACM/IEEE Design Automation Conference* (2001).

[PK00] Paruthi, V., and A. Kuehlmann.: Equivalence Checking Combining a Structural SAT-Solver, BDDs, and Simulation. *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors* 2000, 459– 464.

[PG96] Puri, R., J. Gu.: A BDD SAT Solver for Satisfiability Testing: An Industrial Case Study. *Annals of Mathematics and Artificial Intelligence*, **17** (1996) 315–337.

[SAFS95] Schlipf, J.S., F. Annexstein, J. Franco, and R. Swaminathan.: On finding solutions for extended Horn formulas. *Information Processing Letters*, **54** (1995) 133–137.

[Sch89] Schöning, U.: *Logic for Computer Scientists*. Springer Verlag (1980) 22.

[Sht00] Shtrichman, O.: Tuning SAT Checkers for Bounded Model Checking. *Proceedings of the 12th International Computer Aided Verification Conference* 2000.

[Sta94] Stålmarck, G.: A System for Determining Propositional Logic Theorems by Applying Values and Rules to Triplets that are Generated from a Formula. Swedish Patent No. 467,076 (approved 1992), U.S. Patent No. 5,276,897 (1994), European Patent No. 0403,454 (1995).

[Vel00] Velev, M.N.: Superscalar Suite 1.0. Available from: `http://www.ece.cmu.edu/~mvelev`.

[ZMMM01] Zhang, L., C. Madigan, M. Moskewicz, and S. Malik.: Efficient Conflict Driven Learning in a Boolean Satisfiability Solver. *Proceedings of ICCAD 2001*.