Prime Clauses for Fast Enumeration of Satisfying Assignments to Boolean Circuits[*]

HoonSang Jin, Fabio Somenzi
University of Colorado at Boulder

Abstract

Finding all satisfying assignments of a propositional formula has many applications in the design of hardware and software. An approach to this problem augments a clause-recording propositional satisfiability solver with the ability to add *blocking clauses*, which prevent the solver from visiting the same solution more than once. One generates a blocking clause from a satisfying assignment by taking its complement. In this paper, we present an improved algorithm for finding all satisfying assignments for a generic Boolean circuit. An optimization based on lifting—which generates minimal satisfying assignments—yields prime blocking clauses. Thanks to the primality of the blocking clauses, the derived conflict clauses usually prune both satisfiable and unsatisfiable points at once. The efficiency of our new algorithm is demonstrated by our preliminary results on SAT-based unbounded model checking.

Categories and Subject Descriptors
B.6.3 [**Logic design**]: Design aids—*Verification*
General Terms: Verification, Algorithms
Keywords: SAT, CNF, AllSAT, minimal satisfying assignment

1. Introduction

The satisfiability of a propositional formula is one of the most studied problems in computer science, from both the theoretical and the practical standpoints. The development of efficient satisfiability (SAT) solvers based on the DPLL procedure [3, 2] has led to more problems being tackled by reduction to SAT.

Specifically, the systematic exploration of all satisfying assignments of a propositional formula is important for unbounded SAT-based model checking, for decision procedures for arithmetic constraints, Presburger arithmetic, and various fragments of first order logic, and in manipulations of logic circuits such as complementation and flattening. We call this problem *AllSAT*.

Listing all the solutions to an AllSAT problem requires exponential time in the worst case. All efforts should be made to compute the set of satisfying assignment in a concise form. Normally, the desired format is a DNF formula, which should have as few terms as it is feasible without compromising the speed of the solver. Advances in the design of SAT solvers like non-chronological backtracking, and efficient implementations like those based on two-watched literal schemes, have inspired new approaches to the solution enumeration problem [8].

[*]This work was supported in part by SRC contract 2004-TJ-920.

Conventional SAT solvers aim at computing just one solution, but they can be augmented to enumerate all of them. In principle, to solve AllSAT, it is enough to force the SAT solver to continue the search after getting each satisfying assignment.

In [8], once a satisfying assignment is found then a *blocking clause* is generated by taking its complement. Blocking clauses allow a SAT solver based on conflict clause recording to continue the search without finding the same solution more than once. Optimization techniques are applied to get smaller blocking clauses, since enumerating each satisfying assignment would be inefficient. In [8] the author builds an *auxiliary implication graph* to find a subset of the input assignments sufficient to justify the output of the circuit. Our previous work [6] improves on this approach by performing an additional conflict analysis on each blocking clause.

For lack of space, we refer the reader to [6] for a discussion of related work. We only mention here that recently [5] proposed to represent the set of satisfying assignments as circuits. Circuit cofactors provide an implicit representation of satisfying assignments. Since the authors of [5] implicitly represent the assignments, no minimization techniques are applied. However, the cofactor-based representation may grow large, and is not suitable for applications that expect conjunctions of literals instead.

We propose an efficient algorithm that enumerates all satisfying assignments to a Boolean circuit as conjunctions of literals, so that it can be directly used for the decision of various fragments of first order logic; in manipulations of logic circuits such as complementation and flattening; and in unbounded model checking.

The rest of this paper is organized as follows. Background material is covered in Sect. 2. Section 3 overviews the enumeration algorithm. The minimization algorithm generating prime clauses is presented in Sect. 4; its efficiency is demonstrated by our preliminary results for SAT-based unbounded model checking problems in Sect. 5. Finally, conclusions are drawn in Sect.6.

2. Preliminaries

In this paper we discuss the problem of enumerating the satisfying assignments to a single-output Boolean circuit; that is, enumerating the input assignments for which the output of the circuit is 1. This output is known as the *objective* in what follows.

Most SAT solvers read a propositional formula in Conjunctive Normal Form (CNF). A *literal* is either a variable or its negation. A *clause* is a set of literals; complementary literals are not allowed in one clause. A *CNF formula* is a set of clauses. Boolean circuits are converted to CNF by introducing auxiliary variables for the logic gates or subformulae. The conversion has linear complexity. The variable for the output gate of the circuit is denoted by *obj*. The set of the original input variables to the circuit is denoted by V.

A *prime clause* of a CNF formula F is a clause π that is implied by F and such that every clause obtained from π by removing one or more literals is not implied by F. The collection of all prime clauses of F is called the *complete conjunction* for F.

```
AllSAT($F$, $V$, $obj$) {
    $\hat{F}$ = $F$;
    while (1) {
        while (ChooseNextAssignment()) {
            while (Deduce() == CONFLICT) {
                blevel = AnalyzeConflict();
                if (blevel < 0) return (UNSAT, $\emptyset$);
                else Backtrack(blevel); } }
        $A'$ = MinimizeAssignment($F$, $V$, $A$, $obj$);
        $B = \neg A'$;
        $\hat{F}$ = $\hat{F} \wedge B$; } }
```

Figure 1: AllSAT algorithm

```
MinimizationAndPruning($\hat{F}$, $F$, $V$, $A$, $obj$) {
    BuildAuxiliaryImplicationGraph($F$, $V$, $obj$);
    $A$ = MinimizationBasedOnAntecedent($F$, $obj$);
    $A'$ = MinimizationBasedOnLifting($F$, $A$, $obj$);
    $B = \neg A'$;
    $\hat{F}$ = $\hat{F} \wedge B$;
    if (CheckUIP($F$, $B$) == 0)
        bLevel = GetSafeBacktrackLevel($F$, $B$);
    else
        bLevel = ConflictAnalysis($F$, $B$);
    Backtrack(bLevel); }
```

Figure 2: Algorithm overview

3. Algorithm Overview

Figure 1 shows the basic algorithm to get all satisfying assignments to a propositional formula. The DPLL procedure is extended with the ability to add *blocking clauses* B as in [8]. One starts AllSAT with the given CNF formula F, the set of variables V, and obj. While applying DPLL, if a conflict happens then the resulting conflict clause is added to $\hat{F}$. When a satisfying assignment A is found, minimization is applied to A to yield A'. Then a blocking clause B is generated by taking the complement of the conjunction of the literals in A'. Adding B to $\hat{F}$ prevents the solver from visiting the same solution again. After a blocking clause is added, the search is continued until $\hat{F}$ becomes unsatisfiable.

The formula F is duplicated at the beginning of AllSAT, so that the original formula can be used by MinimizationAndPruning to minimize blocking clauses.

To reduce the literals in A, one may apply a greedy heuristic as in [8] with BuildAuxiliaryImplicationGraph and MinimizationBasedOnAntecedent. While propagating the valuations of the variables in V, if the objective *obj* is satisfied then a sufficient satisfying assignment is identified. Different propagation orders may result in different sufficient sets of variables. The assignments are further minimized by tracing back from *obj* in the implication graph of F. This is done in MinimizationBasedOnAntecedent. However, even backtracing from *obj* does not guarantee minimality of the satisfying assignment.

Figure 3 shows a case in which this method cannot get a minimal assignment. In the example $a = 0$ is a minimal assignment (Part b). If the implication order is b, a, c, the sufficient assignment is found to be $\neg b \wedge \neg a$ (Part a). Only when a is implied first the minimal assignment is detected. To get a minimal satisfying assignment we apply a sophisticated algorithm , which is discussed in Section 4.

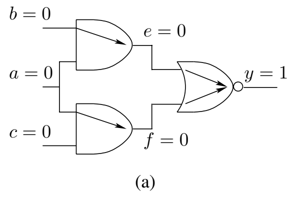

Figure 3: Example of order dependency

After getting the blocking clause B, the algorithm performs conflict analysis on it to derive a proper conflict clause for the modified formula. We skip conflict analysis if B already contains a Unique Implication Point (UIP). If it does, then it is the first UIP under the current implication graph, and conflict analysis would replace it with the next one.

Besides yielding a valid, nontrivial backtracking level, the derived conflict clause is usually more effective at pruning the search space, since it often encompasses both satisfiable and unsatisfiable points. The advantages of this approach are discussed in [6].

4. Minimizing Satisfying Assignments

To get a blocking clause B over the variables in V, Boolean constraint propagation is applied on F instead of $\hat{F}$, which is the original formula, disregarding conflict learned clauses and blocking clauses. The reason is illustrated in Fig. 4.

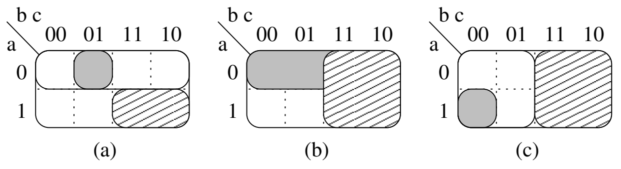

Figure 4: Example of minimization

Suppose we have three variables in our SAT instance, and $a \wedge b$ is the off-set. A satisfying assignment is $\neg a \wedge \neg b \wedge c$: It is expanded to $\neg a$ as shown in (a) and then a new conflict clause $\neg b$ is generated by applying conflict analysis to blocking clause a as in (b). Now we have another satisfying assignment $a \wedge \neg b \wedge \neg c$. If we expand $a \wedge \neg b \wedge \neg c$ with respect to $\hat{F} = F \cup B$, then we cannot get $\neg b$ because of the blocking clause a. But if we expand $a \wedge \neg b \wedge \neg c$ with respect to F, then $\neg b$ is generated and we can terminate. The minimization on F instead of $\hat{F}$ gives us a chance to get further minimization, since the expansion on F may give us a cover made up of prime implicants instead of a disjoint cover.

More sophisticated techniques, such as brute-force lifting [9] or a covering approach [9], can be applied to further minimize the satisfying assignments. These techniques may not be fast enough to be used in the inner loops of the assignment enumeration. In our algorithm, we use lifting (against F) to produce *prime blocking clauses*, but we propose two methods to reduce its cost.

Given a partial satisfying assignment, that is, a set of literals sufficient to justify the objective, one can check the possibility of lifting one of the remaining variables by removing its literal from the assignment, and checking satisfiability of the negation of the formula. If the formula is unsatisfiable, the variable can be lifted. The resulting minimal assignment depends on the order in which the variables are considered and is not necessarily of least-cost. On the other hand, this procedure overcomes the limitations of three-valued simulation of a circuit. In three-valued logic simulation of a circuit, a signal has one of three values: 0 (false), 1 (true), and X (unknown). Consider a multiplexer whose select signal is s and whose data inputs are a and b. Let the implementation be $y = (a \wedge \neg s) \vee (b \wedge s)$. If $a \wedge b \wedge \neg s$ is a satisfying assignment for $y = 1$, then this can be minimized to $a \wedge \neg s$, but only lifting, among the methods discussed so far can reduce it to $a \wedge b$.

The first technique to make lifting more efficient is to use an incremental SAT solver for all the lifting attempts for all satisfying assignments. We interleave the running of two SAT solver instances. One solver is for AllSAT solving and the other is for lifting. All the conflict clauses generated from one lifting process can be used for the next lifting, since the changes between instances are unit literals [4]. As lifting progresses, the SAT instances get easier to solve thanks to the conflict clauses forwarded from previous lifting instances, which generate additional implications, so that we often can conclude SAT or UNSAT immediately without any decision making. The variable scores for lifting are initialized only once, while clause deletion and periodic score decaying are disabled. This considerably enhances speed.

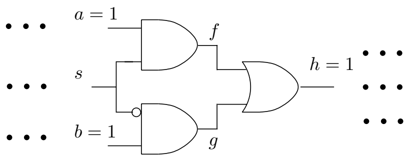

Figure 5: Example of learning

To further reduce the cost of lifting, we restrict the use of lifting by applying it periodically. A sub-optimal variable ordering is one reason why we cannot get a minimal satisfying assignment with greedy minimization. The fact that F is not a complete conjunction is another reason why we cannot get minimal satisfying assignments without lifting. Since lifting can overcome the limitation of an incomplete representation, we try to use it to extract clauses that make F more transparent to implication. Recall that A' is a sufficient set of variables to satisfy *obj*. However, BCP may not imply the value of *obj* because of the limitation of three valued simulation and the incompleteness of F. We can get clauses that help to overcome these limitations by solving SAT on $F' \cup \bigwedge A'$. Since this instance is unsatisfiable, the solver generates conflict clauses. These conflict clauses are helpful to produce more implications from the assignment based on A'. In Figure 5, we show an example of learning. After applying BCP on $\bigwedge A'$ and $\neg obj$, the variables f, g are unassigned. Making decisions on f or g may create a conflict and lead to conflict clause $\neg a \vee \neg b \vee h$; this can create the additional implications later. One may add clauses closing those gaps by analyzing the circuit structure, but in our experiments, this created larger clauses than relying on conflict analysis of the SAT solver.

Finally we also forward useful conflict clauses generated during lifting to $\hat{F}$. By tracing the objectives $\neg$*obj* and $\bigwedge A'$ during implication, we can identify the conflict clauses that are not trivially satisfied under the objective [7]. Figure 6 shows the details of the additional technique based on periodic lifting and additional learning. With periodic lifting we may not find only prime blocking clauses. However, we can bridge the gaps across which implication is impossible. Therefore we can increase the chance of getting close to the minimal satisfying assignments without applying lifting all the time.

The proposed minimization algorithm tends to generate minimal satisfying assignments. If a satisfying assignment is minimal then it abuts some unsatisfying assignments. Hence, conflict analysis on a

```
1   MinimizationBasedOnLifting($F$, $A$, $obj$, $n$) {
2       **if** ($n$ % samplingRate $\neq$ 0) **return** $A$;
3       $F'$ = ($F \setminus obj$) $\cup \neg obj$;
4       $A'$ = $A$;
5       **for each** $l \in A$
6           $F''$ = $F' \cup \neg l \cup \bigwedge(A' \setminus l)$;
7       **if** (SATSOLVE($F''$) == UNSAT)
8           $A'$ = $A' \setminus l$;
9       OBJECTIVEORIENTEDLEARNING($F' \cup \bigwedge A'$);
10      **return** $A'$; }
```

Figure 6: Lifting algorithm

prime blocking clause derives a conflict clause that is usually more effective at pruning the search space, since it usually encompasses both satisfiable and unsatisfiable points. The derived conflict clause provides more guidance to the score-based heuristics that select the decision variables. Therefore, AllSAT finishes sooner with fewer conflicts and blocking clauses, since in our algorithm the decision heuristic is influenced by the blocking clauses as well.

5. Experimental Results

We have implemented the proposed all satisfying assignments algorithm in VIS-2.1 [1, 10]. To show the efficiency of the proposed algorithm on various examples, we implemented a SAT-based unbounded model checker that uses the AX operator as described in [8]. The inputs, that is, the transition relation and the invariant property, are given as Boolean circuits. At every iteration of the AX operation, the frontier is extracted and expressed as a circuit in terms of next state variables. A new objective is created to satisfy the set of states in the frontier. The iteration is continued until convergence.

The model checking examples are selected from the VIS benchmark suite [10]. The examples that could be solved in a few seconds were removed from the set. Some of the selected examples are small, but notoriously difficult for SAT-based methods.

The experiments have been performed on 1.7 GHz Pentium IV with 1 GB of RAM running Linux. We have set the time out limit to 20,000 s.

We compare the proposed algorithm with 4 experimental setups as follows.

- Case A : without pruning, with greedy minimization
- Case B : with pruning, with greedy minimization
- Case C : with pruning, with lifting
- Case D : with pruning, with periodic lifting and learning

The implementations of cases C and D use the incremental SAT, while case B is the algorithm of [6]. Table 1 compares the numbers of pre-image steps that can be completed in a given amount of time; '*' after the number of pre-image steps indicates that convergence was not reached. The improvement due to the pruning algorithm is shown by comparing Cases A and B. We can get an order of magnitude speed-up from B to C because we apply incremental SAT while applying lifting.

The algorithm based on periodic lifting and learning shows its robustness, since Case D either finishes earlier or completes more pre-image steps. We cannot get better result for the 'buf_bug' example than in case B because lifting cannot minimize satisfying assignments further. Therefore all the efforts to minimize assignments are wasted in this case.

Table 1: Performance comparison until reaching timeout

Design	# latches	A PI	A CPU	B PI	B CPU	C PI	C CPU	D PI	D CPU
synch_bakery	22	15	965	15	266	15	41	15	34
solitaireVL	22	17	15728	17	1404	17	633	17	608
s1423	74	4	477	4	340	4	446	4	139
buf_bug	27	18	13891	18	7669	18	13415	18	8449
swap10	40	6*	≥20k	9	≥20k	9	2	9	1
vsa16a	172	4*	≥20k	4*	≥20k	14	619	14	350
bpb	36	7*	≥20k	8*	≥20k	18	1217	18	1103
huff	37	6*	≥20k	12*	≥20k	20	3551	20	3432
eisenberg	30	3*	≥20k	4*	≥20k	18	19036	18	11322
hanoi	54	3*	≥20k	4*	≥20k	5*	≥20k	6*	≥20k
usb_phy	91	7*	≥20k	7*	≥20k	8*	≥20k	9*	≥20k
Ifetchprod	147	4*	≥20k	4*	≥20k	6*	≥20k	6*	≥20k
two	30	16*	≥20k	18*	≥20k	20*	≥20k	22*	≥20k
luckySevenO	30	23*	≥20k	27*	≥20k	33*	≥20k	34*	≥20k
ball	85	6*	≥20k	7*	≥20k	60*	≥20k	67*	≥20k

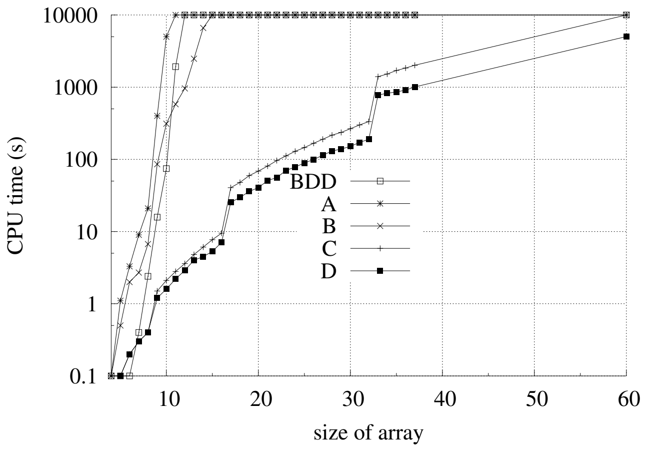

Figure 7: Performance comparison for the *swap* example

We conducted also experiments on the *swap* example of [8]. It consists of n k-bit binary numbers, $x_0, \cdots, x_{n-1}$ with initial values $x_i = i$. The model swaps the values of x_i and $x_{(i-1) \pmod n}$. We check whether all the first two values of the array are distinct. With our proposed minimization heuristic, we can finish up to $n = 35$ within 1000 s on our 1.7 GHz machine. By contrast, the recent algorithm of [5] could only complete $n = 25$ on a substantially faster CPU. We found that the performance of the greedy minimization heuristic heavily depends on the description style of the model and the variable ordering when building the auxiliary implication graph, but the minimization of the blocking clauses improved robustness significantly. As a reference, the curve labeled 'BDD' in the figure shows the result of BDD-based invariant checking done by VIS [1, 10].

6. Conclusions

We have presented a novel algorithm for enumerating the satisfying assignments of a Boolean circuit. The minimization algorithm based on lifting variables generates prime blocking clauses that prune out more search space in concert with the pruning algorithm of [6]. Experimental results show a very significant improvement in the speed of AllSAT solving and a dramatic one in the number of blocking clauses that make up the solution (data not shown for lack of space).

Since our algorithm explicitly enumerates the satisfying assignments, it is well suited for use in decision procedures for various fragments of first order logic, for the optimization of counterexamples, and in the flattening and complementation of logic synthesis.

References

[1] R. K. Brayton et al. VIS: A system for verification and synthesis. In *CAV'96*, pages 428–432. Springer-Verlag, Berlin, 1996. LNCS 1102.

[2] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5:394–397, 1962.

[3] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, July 1960.

[4] N. Eén and N. Sörensson. Temporal induction by incremental SAT solving. First Intl. Workshop on Bounded Model Checking. *ENTCS*, 89(4), 2003.

[5] M. K. Ganai, A. Gupta, and P. Ashar. Efficient SAT-based unbounded symbolic model checking using circuit cofactoring. In *ICCAD*, pages 408–415, Nov. 2004.

[6] H. Jin, H. Han, and F. Somenzi. Efficient conflict analysis for finding all satisfying assignments of a Boolean circuit. In *TACAS'05*, pages 287–300, Apr. 2005. LNCS 3440.

[7] H. Jin and F. Somenzi. An incremental algorithm to check satisfiability for bounded model checking. Second Intl. Workshop on Bounded Model Checking. Boston, MA, July 2004.

[8] K. L. McMillan. Applying SAT methods in unbounded symbolic model checking. In *CAV'02*, pages 250–264. Springer-Verlag, Berlin, July 2002. LNCS 2404.

[9] K. Ravi and F. Somenzi. Minimal assignments for bounded model checking. In *TACAS'04*, pages 31–45, Barcelona, Spain, Mar.-Apr. 2004. LNCS 2988.

[10] URL: http://vlsi.colorado.edu/∼vis.