Efficient Conflict Analysis for Finding All Satisfying Assignments of a Boolean Circuit*

HoonSang Jin, HyoJung Han, and Fabio Somenzi

University of Colorado at Boulder
{Jinh, Hhhan, Fabio}@Colorado.EDU

Abstract. Finding all satisfying assignments of a propositional formula has many applications to the synthesis and verification of hardware and software. An approach to this problem that has recently emerged augments a clause-recording propositional satisfiability solver with the ability to add "blocking clauses." One generates a blocking clause from a satisfying assignment by taking its complement. The resulting clause prevents the solver from visiting the same solution again. Every time a blocking clause is added the search is resumed until the instance becomes unsatisfiable. Various optimization techniques are applied to get smaller blocking clauses, since enumerating each satisfying assignment would be very inefficient.

In this paper, we present an improved algorithm for finding all satisfying assignments for a generic Boolean circuit. Our work is based on a hybrid SAT solver that can apply conflict analysis and implications to both CNF formulae and general circuits. Thanks to this capability, reduction of the blocking clauses can be efficiently performed without altering the solver's state (e.g., its decision stack). This reduces the overhead incurred in resuming the search. Our algorithm performs conflict analysis on the blocking clause to derive a proper conflict clause for the modified formula. Besides yielding a valid, nontrivial backtracking level, the derived conflict clause is usually more effective at pruning the search space, since it may encompass both satisfiable and unsatisfiable points. Another advantage is that the derived conflict clause provides more flexibility in guiding the score-based heuristics that select the decision variables. The efficiency of our new algorithm is demonstrated by our preliminary results on SAT-based unbounded model checking of VIS benchmark models.

1 Introduction

Many applications in computer science rely on the ability to solve large instances of the propositional satisfiability (SAT) problem. Examples include bounded and unbounded model checking [2, 20, 21], equivalence checking [11] and various other forms of automated reasoning [1, 17], test generation, and placement and routing of circuits [23]. While some of these applications only require a yes-no answer, an increasing number of them relies on the solver's ability to provide a proof of unsatisfiability [10, 29], a satisfying assignment that is minimal according to some metric [5, 24], or an enumeration of all satisfying assignments to a propositional formula [20].

* Work supported in part by SRC contract 2004-TJ-920.

Specifically, the systematic exploration of all satisfying assignments 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 the optimization of logic circuits. The problem is computationally hard because listing all the solutions requires exponential time in the worst case. All efforts should be made to present (and compute) the set of satisfying assignment in a concise form. Normally, the desired format is a disjunctive normal form (DNF) formula, which should consist of as few terms as it is feasible without compromising the speed of the solver. Recent advances in the design of SAT solvers like non-chronological backtracking and conflict analysis based on unique implication points, and efficient implementations like those based on two-watched literal schemes have inspired new approaches to the solution enumeration problem as well [20]. In this paper, in particular, we show how to take full advantage of sophisticated conflict analysis techniques to substantially increase both the speed of enumeration and the conciseness of the solution.

Conventional SAT solvers are targeted to computing just one solution, but they can be augmented to get all solutions. We call the problem of finding all satisfying assignments *AllSat*. In principle, to solve AllSat, it is enough to force the SAT solver to continue the search after getting each satisfying assignment.

In previous work [20, 16, 6], once a satisfying assignment is found then a *blocking clause* is generated by taking its complement. Blocking clauses are added to the function being examined to prevent the SAT solver from finding the same solution again. They represent the natural way to force a SAT solver based on conflict clause recording to continue the search. Various optimization techniques are applied to get smaller blocking clauses, since enumerating each satisfying assignment would be very inefficient. In applications like unbounded model checking [20], one seeks to enumerate the assignments to a propositional formula originally given in the form of a circuit. The translation of this circuit to conjunctive normal form (CNF) introduces auxiliary variables whose values are determined by those of the inputs of the circuit. The solutions that are enumerated should only be in terms of these input variables, and the minimization of the blocking clauses should take this feature into account. One way to achieve this objective is to use a so-called *auxiliary implication graph* to determine a subset of the input assignments sufficient to justify the output of the circuit.

The approach of [16] uses an external two-level minimization program to get a minimized form for sets of satisfying assignments instead of finding a minimized blocking clause internally on the fly. Every time a solution is identified then it is saved in DNF in addition to adding its negation as a blocking clause. The accumulated DNF is periodically fed to a two-level minimizer. All recent advances in propositional SAT [22, 9, 15] based on DPLL [8, 7] can be adopted to enhance performance of these AllSat methods, since they use exactly the same SAT algorithm except for the addition of the blocking clauses as additional constraints.

In [19, 12], the authors point out that the size of the instance may be increased significantly by the addition of the blocking clauses. Consequently, the speed of finding one solution is decreased because of the time spent in implications for those blocking clauses. They propose to save solutions in a decision tree by restricting non-chronological

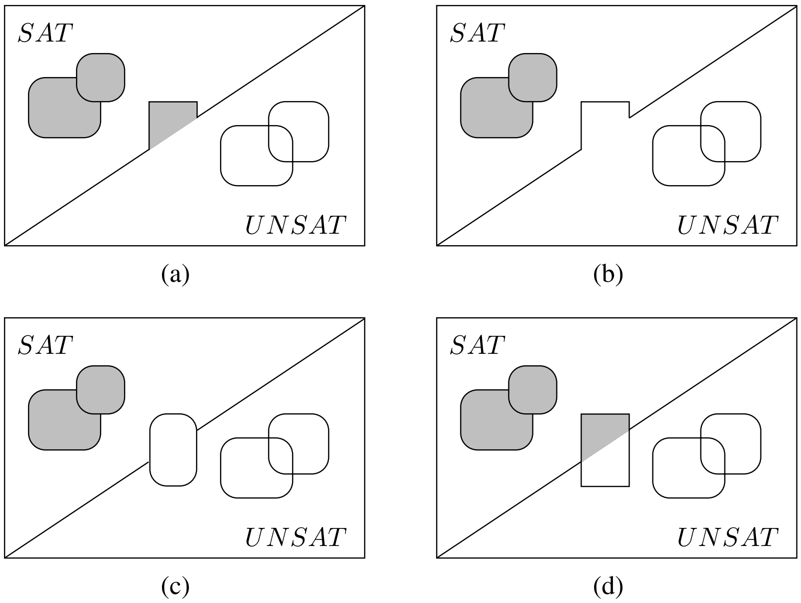

Fig. 1. Illustration of AllSat solving

backtracking. The decision heuristic also is restricted to increase the chance of saving solutions into the decision tree.

A simple algorithm for AllSat based on adding blocking clauses is illustrated in Fig. 1. For a given SAT instance, the search space can be divided into SAT and UNSAT subspaces. In the figure, the filled rectangles represent the blocking clauses created from satisfying assignments and the unfilled rectangles represent the conflict clauses generated from conflict analysis. In conventional SAT solving, if the satisfying assignment is identified on the SAT side, then the search is terminated. By contrast, in AllSat, the search is continued to cover all SAT and UNSAT points. When all the search space is covered by blocking clauses and conflict clauses then AllSat solving is finished. Minimization techniques can enlarge the filled rectangles as in [20, 16, 6]. Since large rectangles can prune large parts of the search space, these minimization techniques are beneficial.

Suppose that while solving AllSat, we have a blocking clause close to the UNSAT side, as illustrated in Part (a) of the figure. By adding the blocking clause, the corresponding satisfying assignments are moved to the UNSAT side as in (b). In the future, the solver may find a conflict such that the conflict-learned clause prunes this part of search space as in (c). Since finding blocking clauses and conflict analysis can only be applied on the SAT and UNSAT sides, respectively, these conventional solvers cannot apply powerful pruning techniques on both SAT and UNSAT side simultaneously as in (d). That is, they cannot add clauses that prevent the future exploration of both SAT and UNSAT points of the search space. In this paper we propose an efficient conflict analysis that removes this limitation. This technique improves the effectiveness of non-chronological backtracking, and of the heuristic that chooses the decision variables. The main idea is to regard the blocking clause as a conflicting clause for the updated function, and to generate a conflict clause from it. An important issue is that this additional conflict analysis requires, for an efficient implementation, that the state of the SAT solver be preserved across the computation of the blocking clause. In our solver this is very naturally accomplished by letting the input circuit be represented in the form of an And-Inverter Graph (AIG), while conflict and blocking clauses are kept in CNF.

The rest of this paper is organized as follows. Background material is covered in Section 2. Section 3 presents the new AllSat algorithm based on a hybrid representation, which consists of AIG and CNF. The efficiency of our new algorithm is demonstrated by our preliminary results for SAT-based unbounded model checking problems in Section 4. Finally, we draw conclusions in Sect.5.

2 Preliminaries

Most SAT solvers read a propositional formula in CNF. Boolean circuits, which are often encountered in design automation applications of SAT, are converted to CNF by introducing auxiliary variables for the logic gates or subformulae. The conversion to CNF has linear complexity, and is therefore efficient. Recently, however, there have been proposals for SAT solvers that combine the strengths of different representations, including circuits, CNF formulae, and Binary Decision Diagram (BDD [4]) [18, 14].

In this paper we rely on both CNF and AIGs to solve the AllSat problem. An *And-Inverter Graph* (AIG) is a Boolean circuit such that each internal node ν has exactly two predecessors, and if the predecessor variables are v_1 and v_2, its function $\phi(\nu)$ is one of $v_1 \wedge v_2$, $v_1 \wedge \neg v_2$, $\neg v_1 \wedge v_2$, and $\neg v_1 \wedge \neg v_2$. A *Conjunctive Normal Form* (CNF) is a set of *clauses*; each clause is a set of *literals*; each literal is either a variable or its complement. The function of a clause is the disjunction of its literals, and the function of a CNF formula is the conjunction of its clauses.

Figure 2 shows the pseudocode for the DPLL procedure. Procedure ChooseNextAssignment checks the implication queue. If the queue is empty, the procedure makes a *decision*: it chooses one unassigned variable and a value for it, and adds the assignment

```
1  DPLL() {
2      while (ChooseNextAssignment()) {
3          while (Deduce() == CONFLICT) {
4              blevel = AnalyzeConflict();
5              if (blevel ≤ 0) return UNSATISFIABLE;
6              else Backtrack(blevel);
7          }
8      }
9      return SATISFIABLE;
10 }
```

Fig. 2. DPLL algorithm

```
AllSat() {
    while (1) {
        while (ChooseNextAssignment()) {
            while (Deduce() == CONFLICT) {
                blevel = AnalyzeConflict();
                if (blevel ≤ 0) return UNSATISFIABLE;
                else Backtrack(blevel);
            }
        }
        FindAndAddBlockingClause();
        Backtrack();
    }
}
```

Fig. 3. AllSat algorithm

to the implication queue. If none can be found, it returns false. This causes DPLL to return an affirmative answer, because the assignment to the variables is complete.

If a new assignment has been chosen, its implications are added by Deduce to the queue. Efficient computation of implications for clauses is discussed in [25, 27, 22, 9]; implications in AIGs are described in [18];

If the implications yield a conflict, AnalyzeConflict() is launched. Conflict analysis relies on the (implicit) construction of an *implication graph*. Each literal in the conflicting clause has been assigned at some level either by a decision, or by an implication. If there are multiple literals from the current decision level, at least one of them is implied. Conflict analysis locates the source of that implication—it may be a clause or an AIG node—and extends the implication graph by adding arcs from the antecedents of the implication to the consequent. This process continues until there is exactly one assignment for the current level among the leaves of the tree. The disjunction of the negation of the leaf assignments gives then the conflict clause. The highest level of the assignments in the conflict clause, excluding the current one, is the backtracking level. The single assignment at the current level is known as first *Unique Implication Point* (UIP). Conflict clauses based on the first UIP have been empirically found to work well [28].

Conflict analysis produces two important results. The first is a clause implied by the given circuit and objectives. This *conflict clause* is added to the clauses of the circuit. Termination relies on it, because it causes the search to continue in a different direction. The second result of conflict analysis is the *backtracking level*: Each assignment to a variable has a *level* that starts from 0 and increases with each new decision. When a conflict is detected, the algorithm determines the lowest level at which a decision was made that eventually caused the conflict. The search for a satisfying assignment resumes from this level by deleting all assignments made at higher levels. This *non-chronological backtracking* allows the decision procedure to ignore inconsequential decisions that have provably no part in the conflict being analyzed.

Figure 3 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* as in [20, 6]. One generates a blocking clause from a satisfying assignment by taking the complement of the conjunction of all the literals in the assignment. Procedure FIND-ANDADDBLOCKINGCLAUSE() finds a blocking clause and adds it to the database. The resulting clause prevents the solver from visiting the same solution again. Every time a blocking clause is added, the search is resumed at some safe backtracking level until the instance becomes unsatisfiable.

3 Algorithm

In the technique we propose, an AIG is used to represent the Boolean circuit whose satisfying assignments must be enumerated, while the result of a conflict analysis is represented as one clause. In our framework, conflict analysis and implications can be applied to both CNF formulae and AIGs. Figure 4 shows the pseudocode for the proposed algorithm. The naive algorithm of Figure 3 can be improved by replacing lines 10 and 11 with the procedure in Figure 4.

In the algorithm description, C is a Boolean circuit in the form of an AIG, which is given as an input together with obj; obj is the *objective*, which is a node of C. We want to find the all the assignments over V that satisfy obj. F is the formula resulting from conjoining C with the conflict clauses and the blocking clauses generated while solving the AllSat problem. Therefore, initially F is C and when F becomes 0 AllSat is completed.

Procedure BLOCKINGCLAUSEANALYSIS is called when a satisfying assignment is found in F. To get a blocking clause B over the variables in V, Boolean constraint propagation is applied on C, which is the original Boolean circuit, disregarding conflict learned clauses and blocking clauses. This is to get a smaller assignment from this analysis. Figure 5 shows the reason why C is used for finding minimized assignments. Suppose we have 4 variables in our SAT instance, and $a \wedge c$ is the off-set. $\neg a \wedge b \wedge \neg c \wedge d$ is a satisfying assignment that was detected at an earlier stage. It is possible to get such an assignment if we use a heuristic minimization algorithm, but with small changes,

```
BlockingClauseAnalysis($F, C, A, V, obj$) {
    $B = \emptyset$;
    **for each** $v \in V$ {
        $B = B \cup v$;
        BCPONCIRCUIT($C, v, A(v)$);
        **if** (VALUE($obj$) == $A(obj)$) **break** ;
    }
    MINIMIZATIONBASEDONANTECEDENT($C, B, obj$);
    ADDBLOCKINGCLAUSE($F, \neg B$);
    **if** (CHECKUIP($F, B$) == 0)
        bLevel = CONFLICTANALYSIS($F, \neg B$);
    **else**
        bLevel = GETSAFEBACKTRACKLEVEL($F, \neg B$);
    BACKTRACK(bLevel);
}
```

Fig. 4. Blocking clause analysis algorithm

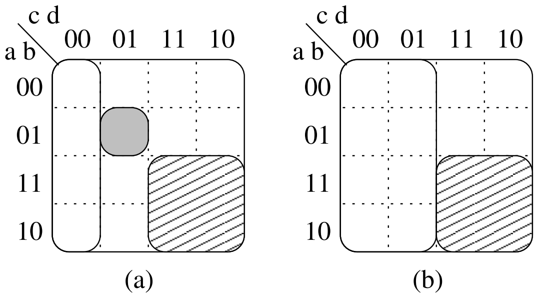

Fig. 5. Example of minimization

the example could be adapted to the case in which exact minimization is applied. Now suppose that satisfying assignment $\neg c \wedge \neg d$ is found. If we try to expand $\neg c \wedge \neg d$ with respect to F, which is $C \wedge (a \vee \neg b \vee c \vee \neg d)$, it cannot be expanded further as in (a). But if we apply minimization with respect to C, it can be expanded to $\neg c$ as in (b). Therefore heuristic minimization on C instead of F will give us a chance to get further minimization, since the expansion on C may give us a cover made up of prime implicants instead of a disjoint cover. In our framework, implications on either the AIG or the CNF clauses can be disabled. Since the conflict clauses and blocking clauses are saved as clauses, the implication on C can be done without extra effort by enabling the implication on AIGs only.

While applying Boolean constraints propagation (BCP) based on valuations of the variables in V, if the objective obj is satisfied then a sufficient satisfying assignment is identified. There is still room to improve because different orders of application of Boolean constraint propagation may result in different sufficient sets of variables. The assignments are further minimized by checking the implication graph C. Figure 6 (a) shows a small example that illustrates order dependency. If we check if y is implied while applying BCP with the order of a, b, c, d then $\neg a \wedge \neg b \wedge c \wedge d$ is identified as a satisfying assignment. If we check if y is implied while applying BCP with the order of c, d, a, b then $c \wedge d$ is a sufficient assignment. To reduce this inefficiency caused by the order of BCP, we traverse the implication graph after the implication on y has been obtained. Even though we apply BCP with the order of a, b, c, d, $c \wedge d$ is detected as sufficient assignment by traversing implication graph on C even with inefficient order of BCP. This is done in MINIMIZATIONBASEDONANTECEDENT. This procedure is similar to the techniques used in [20, 13]. It should be noted that this method does not guarantee minimality of satisfying assignment. Figure 6 (b) shows a case in which we may not get a minimal assignment with this method. In this example $a = 0$ is a minimal assignment. If the implication order is b, a, c, the sufficient assignment is found to be $\neg b \wedge \neg a$. Only when a is implied first the minimal assignment is detected.

Thanks to our solver's hybrid capability, reduction of the blocking clauses can be efficiently performed on C without altering the solver's state (e.g., its decision stack for F and the two-watched literal lists). Therefore the blocking clause is added to F, immediately generating a conflict on F. If a blocking clause has only one variable assignment at the maximum decision level of the implication graph of F, then it already

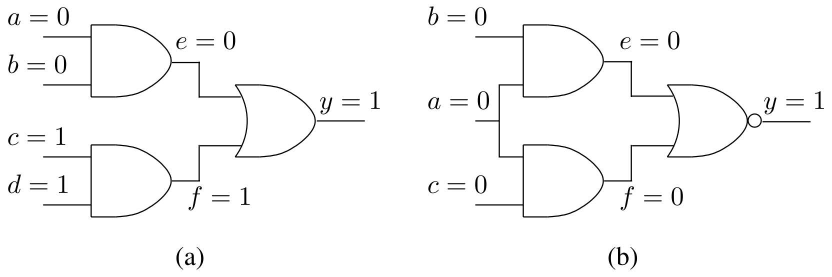

Fig. 6. Example of order dependency

has a unique implication point in it. Otherwise the conflict analysis is applied to get a conflict-learned clause.

Since the solver's state is not altered during the minimization procedure as opposed to implementations based on CNF solver [20, 6], this eliminates a reason for restarting the solver every time a blocking clause is added. One can avoid the restart by duplicating the whole solver data base so as not to alter the solver's state. However, it is not efficient to find an appropriate backtracking level where the search should be resumed based on blocking clause. Our algorithm, on the other hand, performs conflict analysis on the blocking clause to derive a proper conflict clause for the modified formula. 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 as shown in Figure 1. It is also unrestricted in the sense that both input and internal variables of the Boolean circuit may appear in it.

A final advantage of our analysis is that the derived conflict clause provides more flexibility in guiding the score-based heuristics that select the decision variables. If the blocking clauses are used to update the variable scores, the scores of the variables in V will unduly increase, since the variables in the blocking clauses are restricted to variables in that set. This will cause the solver to make decisions almost only on those variables. The result may be beneficial for the pruning of the SAT part of the search space, but not for the UNSAT part. The conflict clauses generated by the proposed algorithm still contain a lot of variables from V. The variables implied earlier than at the current decision level in a conflicting clause are immediately added to the conflict learned clause. Since a conflicting clause—in this case, the blocking clause that was added to F—consists of those variables, the resulting conflict clause still contains a lot of variables in V. To avoid increasing the score of those variables, we use the Deepest Variable Hiking (DVH) decision heuristics [15]. Boosting only the score of the most recent decision variable among the literals in the conflict-learned clause results in a more balanced approach at letting the blocking clauses influence the search direction.

With proof identical to that of Proposition 1 in [29] one shows that the improved All-Sat procedure that performs conflict analysis on the blocking clauses does not require the addition of either blocking clauses or conflict clauses to F to guarantee termination. However, as in standard SAT solvers, these additional clauses may prove useful by causing implications during BCP. Many tradeoffs are possible. In our current implementation we keep both blocking and conflict clauses.

We show that the conflict clauses generated by the proposed algorithm never block other satisfying assignment as follows.

Lemma 1. *For formula F, the conflict learned clause γ generated by conflict analysis on blocking clause β is implied by $F \wedge \beta$.*

Proof. Since the conflict occurred after adding blocking clause to F, $F \wedge \beta \rightarrow \gamma$. □

Theorem 1. *For formula F, the conflict learned clause γ generated by conflict analysis on blocking clause β never blocks satisfying assignments not in β.*

Proof. Since $F \wedge \beta$ does not block any satisfying assignments except $\neg\beta$, γ cannot block other satisfying assignments by Lemma 1. □

All satisfying assignments produced by the algorithms are satisfying assignments of C, because at all times $F \rightarrow C$. In summary, the AllSat algorithm terminates after having enumerated all satisfying assignments of C, and is therefore correct.

4 Experimental Results

We have implemented the proposed all satisfying assignments algorithm in VIS-2.1 [3, 26]. 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 [20].

Our algorithm should already be faster than the one described in [20] because of the hybrid capability and minimization technique discussed in Section 3. Therefore we

Table 1. Performance comparison for reaching a given pre-image step

Design	# latches	pre-image steps	With CA		Without	
			CPU time	# blocking	CPU time	# blocking
synch_bakery	22	49	676.7	37651	2398.9	67280
itc-b07	45	20	628.0	492	1607.2	870
solitaireVL	22	12*	2155.2	17754	15698.5	44693
heap	24	4*	847.4	34815	3028.4	78654
eight	27	2*	487.7	99992	1939.5	144563
buf_bug	27	8*	3128.9	26343	4160.6	39266
swap	27	5*	1893.7	16224	15808.3	24234
two	30	15*	3414.7	28866	10537.5	48961
luckySevenONE	30	24*	6846.2	40963	17657.0	70465
cube	32	10*	4498.9	46214	8013.5	47018
bpb	36	6*	103.3	5648	3823.8	79946
huff	37	9*	7787.1	100855	13441.5	113222
ball	86	6*	46.8	26659	176.8	52772
s1423	74	2*	2039.0	64097	7153.7	78928
Ifetchprod	147	3*	233.1	47103	716.5	57278

Table 2. Performance comparison until reaching timeout

Design	# latches	With CA		Without	
		pre-image steps	CPU time	pre-image steps	CPU time
vsa16a	172	2	1086	0*	≥20000
swap	27	6	3458	4*	≥20000
solitaireVL	22	46	6695	13*	≥20000
eight	27	4*	≥20000	3*	≥20000
two	30	18*	≥20000	16*	≥20000
luckySevenONE	30	27*	≥20000	25*	≥20000
cube	32	11*	≥20000	11*	≥20000
heap	32	6*	≥20000	5*	≥20000
huff	37	17*	≥20000	10*	≥20000
s1423	74	4*	≥20000	2*	≥20000
ball	86	8*	≥20000	7*	≥20000

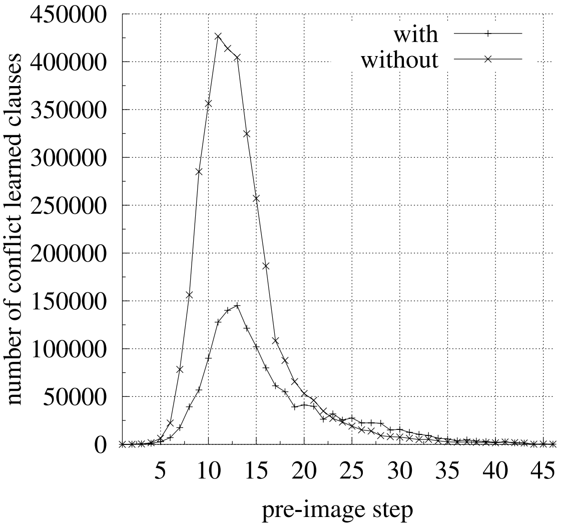

Fig. 7. Number of conflict learned clauses

compare the proposed algorithm with and without conflict analysis on blocking clauses on our hybrid solver rather than comparing with a CNF SAT based implementation. The experimental setup is as follows. 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 [26]. The examples that could be solved in a few seconds were removed from the set. There is not

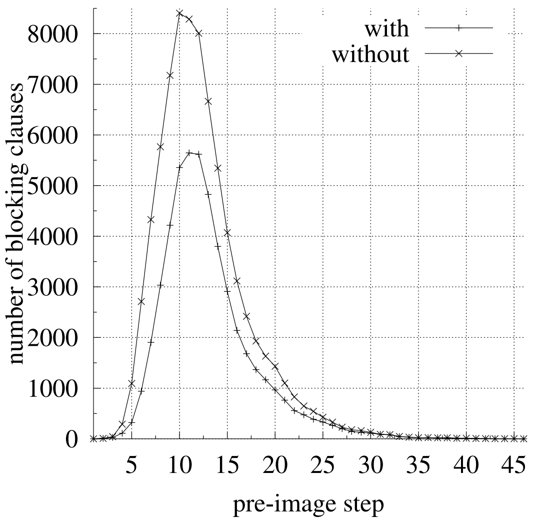

Fig. 8. Number of blocking clauses

direct correlation between the size of the model and its difficulty for model checking. In several of the large examples we considered, the cones of influence of the given properties are small so that the pre-images based on AllSat are trivial. By contrast, most 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.

Table 1 compares the CPU time spent and the number of blocking clauses generated until the same number of pre-image steps is reached. The number is the one that the slower method can complete within the allotted time. The column labeled 'With CA' shows the performance of the proposed algorithm that applies conflict analysis on the blocking clauses. The column labeled 'Without' shows the performance of AllSat without having conflict analysis on the blocking clauses. All other features described in Sect. 3 are still applied for the results shown in this column. The comparison of the two sets of results highlights consistent speed-up (up to 20 times). A '*' after the number of pre-image steps signals that the performed pre-image steps did not suffice for convergence. Even though we have consistent speed-up, some examples benefit more than others from the improved algorithm. We conjecture that this is mainly due to the distribution of satisfying and unsatisfying assignments over the search space.

Table 2 compares the numbers of pre-image steps that can be completed in a given amount of time. The proposed algorithm never finishes fewer pre-image steps. Again, a '*' after the number of pre-image steps indicates that convergence was not reached. For example, 'vsa16a' converges in two pre-image steps with the proposed conflict analysis, while it times out in the first iteration without the proposed conflict analysis.

Figures 7 and 8 show the numbers of conflict learned clauses and the numbers of blocking clauses of the 'solitaireVL' example. Figure 7 supports our claim that the proposed algorithm generates conflict clauses encompassing both SAT and UNSAT points, since it shows large reductions in conflict clauses. We also get smaller numbers of blocking clauses as shown in Figure 8. This seems to confirm our conjecture that the DVH decision heuristic tends to make better choices when the scores are updated based on the conflict clauses generated from the blocking clauses.

5 Conclusions

We have presented a novel conflict analysis on blocking clauses to accelerate the search of all satisfying assignments to a Boolean circuit. The conflict clauses generated by the proposed algorithm often cover both the SAT and the UNSAT side of the search space. This helps in preventing future conflicts. Moreover, we can prune larger parts of the search space, which helps AllSat finish sooner with fewer conflicts. Since in the proposed algorithm the decision heuristic is influenced by the blocking clauses as well, the search is directed in such a way that even larger conflict clauses spanning both the original UNSAT space and the identified SAT points are obtained through the standard conflict analysis. Experimental results show a significant improvement in the speed of AllSat solving and in the number of blocking clauses that make up the solution.

An improved solver for AllSat will benefit many applications that use it directly, like the SAT-based model checker we used to validate our implementation, or that use it as the basis of another algorithm. Examples of the latter that we plan to investigate are decision procedures for various fragments of first order logic.

Acknowledgment

We thank Ken McMillan for clarifying to us the details of his implementation.

References

[1] G. Audemard, P. Bertoli, A. Cimatti, A. Kornilowicz, and R. Sebastiani. A SAT based approach for solving formulas over boolean and linear mathematical propositions. In *International Conference on Automated Deduction*, July 2002.

[2] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Fifth International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'99)*, pages 193–207, Amsterdam, The Netherlands, March 1999. LNCS 1579.

[3] R. K. Brayton et al. VIS: A system for verification and synthesis. In T. Henzinger and R. Alur, editors, *Eighth Conference on Computer Aided Verification (CAV'96)*, pages 428–432. Springer-Verlag, Rutgers University, 1996. LNCS 1102.

[4] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[5] D. Chai and A. Kuehlmann. A fast pseudo-Boolean constraint solver. In *Proceedings of the Design Automation Conference*, pages 830–835, Anaheim, CA, June 2003.

[6] P. Chauhan, E. M. Clarke, and D. Kroening. Using SAT based image computation for reachability analysis. In *Technical Report CMU-CS-03-151*, 2003.

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

[8] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the Association for Computing Machinery*, 7(3):201–215, July 1960.
[9] E. Goldberg and Y. Novikov. BerkMin: A fast and robust SAT-solver. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 142–149, Paris, France, March 2002.
[10] E. Goldberg and Y. Novikov. Verification of proofs of unsatisfiability for CNF formulas. In *Design, Automation and Test in Europe (DATE'03)*, pages 886–891, Munich, Germany, March 2003.
[11] E. Goldberg, M. Prasad, and R. Brayton. Using SAT for combinational equivalence checking. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 114–121, June 2001.
[12] O. Grumberg, A. Schuster, and A. Yadgar. Memory efficient all-solutions SAT solver and its application for reachability analysis. In *Proceedings of FMCAD*, Austin, TX, 2004. To appear.
[13] M. K. Iyer, G. Parthasarathy, and K.-T. Cheng. SATORI – a fast sequential SAT engine for circuits. In *Proceedings of the International Conference on Computer-Aided Design*, pages 320–325, San Jose, CA, November 2003.
[14] H. Jin, M. Awedh, and F. Somenzi. CirCUs: A satisfiability solver geared towards bounded model checking. In R. Alur and D. Peled, editors, *Sixteenth Conference on Computer Aided Verification (CAV'04)*. Springer-Verlag, Berlin, July 2004. To appear.
[15] H. Jin and F. Somenzi. CirCUs: A hybrid satisfiability solver. In *International Conference on Theory and Applications of Satisfiability Testing (SAT 2004)*, Vancouver, Canada, May 2004.
[16] H.-J. Kang and I.-C. Park. SAT-based unbounded symbolic model checking. In *Proceedings of the Design Automation Conference*, pages 840–843, Anaheim, CA, June 2003.
[17] D. Kroening, J. Ouaknine, S. A. Seshia, , and O. Strichman. Abstraction-based satisfiability solving of Presburger arithmetic. In *Sixteenth Conference on Computer Aided Verification (CAV'04)*, pages 308–320, July 2004.
[18] A. Kuehlmann, M. K. Ganai, and V. Paruthi. Circuit-based Boolean reasoning. In *Proceedings of the Design Automation Conference*, pages 232–237, Las Vegas, NV, June 2001.
[19] B. Li, M. S. Hsiao, and S. Sheng. A novel SAT all-solutions for efficient preimage computation. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 380–384, March 2004.
[20] K. L. McMillan. Applying SAT methods in unbounded symbolic model checking. In E. Brinksma and K. G. Larsen, editors, *Fourteenth Conference on Computer Aided Verification (CAV'02)*, pages 250–264. Springer-Verlag, Berlin, July 2002. LNCS 2404.
[21] K. L. McMillan. Interpolation and SAT-based model checking. In W. A. Hunt, Jr. and F. Somenzi, editors, *Fifteenth Conference on Computer Aided Verification (CAV'03)*, pages 1–13. Springer-Verlag, Berlin, July 2003. LNCS 2725.
[22] M. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the Design Automation Conference*, pages 530–535, Las Vegas, NV, June 2001.
[23] G. Nam, F. Aloul, K. Sakallah, and R. Rutenbar. A comparative study of two boolean formulations of FPGA detailed routing constaraints. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, pages 688–696, June 2004.
[24] K. Ravi and F. Somenzi. Minimal assignments for bounded model checking. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'04)*, pages 31–45, Barcelona, Spain, March-April 2004. LNCS 2988.
[25] J. P. M. Silva and K. A. Sakallah. Grasp—a new search algorithm for satisfiability. In *Proceedings of the International Conference on Computer-Aided Design*, pages 220–227, San Jose, CA, November 1996.

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

[27] H. Zhang. SATO: An efficient propositional prover. In *Proceedings of the International Conference on Automated Deduction*, pages 272–275, July 1997. LNAI 1249.

[28] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik. Efficient conflict driven learning in Boolean satisfiability solver. In *Proceedings of the International Conference on Computer-Aided Design*, pages 279–285, San Jose, CA, November 2001.

[29] L. Zhang and S. Malik. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *Design, Automation and Test in Europe (DATE'03)*, pages 880–885, Munich, Germany, March 2003.