Alembic: An Efficient Algorithm for CNF Preprocessing*

Hyojung Han, Fabio Somenzi
University of Colorado at Boulder

Abstract

Satisfiability (SAT) solvers often benefit from a preprocessing of the formula to be decided. For formulae in conjunctive normal form (CNF), subsumed clauses may be removed or partial resolution may be applied. Preprocessing aims at simplifying the formula and at increasing the deductive power of the solver. These two objectives are sometimes competing. We present a new algorithm that combines simplification and increase of deductive power and we show its effectiveness in speeding up SAT solvers.

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

1. Introduction

The last decade has seen great advances in the performance of satisfiability solvers for propositional logic, in particular those based on the DPLL procedure [3, 2, 9, 11, 5]. These solvers have evolved in symbiotic relationship with many EDA applications including model checking, logic synthesis, testing, and timing analysis.

Preprocessing of the input has proved effective in speeding up SAT solvers [4, 14, 18] and, in general, in simplifying formulae [16]. For the common case of Conjunctive Normal Form (CNF), clauses that are subsumed by other clauses slow down the implication process, but do not help the solver in pruning the search space. Therefore, a common goal of preprocessing is to remove them. In addition, resolution may be applied selectively to eliminate literals from clauses. For instance, $a \vee b$ and $a \vee \neg b \vee c$ give a resolvent $a \vee c$ that subsumes the latter; this is called *self subsumption* in [4]. Resolution can also be applied to eliminate variables from the formula. If, for example, $a \vee b$, $\neg b \vee c$ and $\neg b \vee d$ are the only clauses containing b, then they can be replaced by $a \vee c$ and $a \vee d$ while guaranteeing *equisatisfiability*. Since variable substitution may increase the number of clauses, it is usually applied with restraint.

A problem related to preprocessing of a CNF formula is the preprocessing of conflict clauses in an incremental SAT solver. An incremental solver is given a sequence of SAT instances and tries to use clauses learned in earlier instances to expedite the solution of later instances. If each instance is obtained from the previous by addition of new clauses, all clauses learned by the solver can be *forwarded* to the new instance. However, in the general case, clauses must be validated before they can be forwarded. In [7], a process called *distillation* was proposed, which forwards a clause derived from a previously learned clause γ only if asserting the negation of γ causes a conflict in the new instance.

In this paper we apply the assertion of the negation of a clause to preprocessing the original clauses of a CNF formula. We introduce the notion of *deduction power* of a CNF formula to characterize the transformations that help the SAT solver prune more of the search space. For instance, a set of clauses is *redundant* if a proper subset represents the same function. Though subsumed clauses are redundant, and the cost of many SAT solver operations decreases with a smaller formula, not all redundant clauses can be removed without negative effect on deductive power. In fact, the conflict clauses learned by SAT solvers are by definition redundant, but we show that they always improve the deductive power of a CNF formula.

Unlike previous approaches, our distillation procedure may replace a clause with the resolvent of two or more existing clauses without explicitly deriving any such resolvents in advance.

Our experimental results show that our approach allows considerably more simplification than the one of [4], even though it is sometimes slower. Its benefits are more conspicuous for hard unsatisfiable instances, especially those arising from verification problems, for which the SAT solver is sometimes sped up by orders of magnitude by our distillation-based preprocessing.

The rest of this paper is organized as follows. In Sect. 2 we introduce and characterize the notion of deductive power of a CNF formula and describe the principles and results behind our distillation-based approach. Section 3 presents the details of the algorithm and Sect. 4 reports results from a prototype implementation. We draw conclusions and outline future work in Sect. 5.

2. Deductive Power of a CNF Formula

In this paper we assume that the input to the SAT solver is a formula in Conjunctive Normal Form (CNF). A CNF formula is a set of *clauses*; each clause is a set of *literals*; each literal is either a variable or its negation. The function of a clause is the disjunction of its literals, and the function of a CNF formula is the conjunction of its clauses. For instance, the set of clauses

$$\{\{\neg a, c\}, \{\neg b, c\}, \{\neg a, \neg c, d\}, \{\neg b, \neg c, \neg d\}\}$$

corresponds to the following propositional formula:

$$(\neg a \vee c) \wedge (\neg b \vee c) \wedge (\neg a \vee \neg c \vee d) \wedge (\neg b \vee \neg c \vee \neg d).$$

Clause γ_1 subsumes clause γ_2 if $\gamma_1 \subseteq \gamma_2$. Given $\gamma_1 = \gamma_1' \cup \{l\}$ and $\gamma_2 = \gamma_2' \cup \{\neg l\}$, the *resolvent* of γ_1 and γ_2 is $\gamma_1' \cup \gamma_2'$ and is implied by $\{\gamma_1, \gamma_2\}$. An *assignment* for CNF formula F over the set of variables V is a mapping from V to $\{\mathsf{true}, \mathsf{false}\}$. A *partial* assignment maps a subset of V. A satisfying assignment for CNF formula F is one that causes F to evaluate to true. We represent assignments by sets of *unit* clauses, that is, clauses containing exactly one literal. For instance, the partial assignment that sets a and b to true and d to false is written $\{a, b, \neg d\}$ or, interchangeably, $a \wedge b \wedge \neg d$. Given CNF formulae F_1 and F_2 over variables V, F_1 *implies* F_2, written $F_1 \rightarrow F_2$, if all the assignments to V that satisfy F_1 satisfy F_2; F_1 and F_2 are *equivalent* if $F_1 \rightarrow F_2$ and

*This work was supported in part by SRC contract 2006-TJ-1365.

DAC 2007, June 4–8, 2007, San Diego, California, USA.

```
1  DPLL() {
2    while (CHOOSENEXTASSIGNMENT() == FOUND)
3      while (DEDUCE() == CONFLICT) {
4        blevel = ANALYZECONFLICT();
5        if (blevel < 0) return UNSATISFIABLE;
6        else BACKTRACK(blevel);
7      }
8    return SATISFIABLE;
9  }
```

Figure 1: DPLL algorithm.

$F_2 \rightarrow F_1$. A clause γ is *asserting* under assignment A if all its literals except one (the asserted literal) are false. We say that an asserting clause is an *antecedent* of its asserted literal.

Many successful SAT solvers are based on the DPLL procedure, whose modern incarnations are described by the pseudocode of Fig. 1. The solver maintains a current partial assignment that is extended until it either becomes a total satisfying assignment, or becomes conflicting. In the latter case, the solver analyzes the conflict and *backtracks* accordingly. Conflict analysis leads to learning a *conflict clause*, that is, a clause implied by the given clauses that will prevent the solver from trying a set of conflicting assignments (including the one that led to its discovery). Conflict clauses are usually *recorded* to speed up subsequent search.

A full description of the DPLL procedure is beyond the scope of this paper: The reader is referred to the cited work, especially [9, 17, 8]. We note, however, that among the operations performed by a DPLL-based SAT solver, deduction plays a key role in boosting efficiency by finding what literals are implied by the current partial assignment. Deduction is usually based on *modus ponens*:

$$\frac{P, \neg P \vee Q}{Q}. \tag{1}$$

Specifically, SAT solvers apply modus ponens to asserting clauses to deduce the truth of asserted literals: Given a clause $\{l_1, \ldots, l_n\}$ and a partial assignment $\{\neg l_1, \ldots, \neg l_{n-1}\}$, modus ponens deduces l_n. We denote the deductive system that employs this rule alone by $\mathcal{A}$ and we write $F \vdash_{\mathcal{A}} l$ if the truth of l can be established by repeatedly applying modus ponens to asserting clauses in F. An assignment A to the variables of F is *conflicting* if there exists a clause of F that evaluates to false under A, or there is a literal l in F such that $F \cup A \vdash l$ and $F \cup A \vdash \neg l$. Clearly, $\mathcal{A}$ is sound but not complete. For instance, it is not sufficient to deduce that

$$(a \vee b) \wedge (a \vee \neg b) \wedge (\neg a \vee c) \wedge (\neg a \vee \neg c)$$

is unsatisfiable. By contrast, $\mathcal{A}$ would deduce unsatisfiability of

$$(a \vee b) \wedge (a \vee \neg b) \wedge \neg a.$$

A derivation $F \cup A \vdash_{\mathcal{A}} l$ is conveniently represented by its *implication hypergraph*. An implication hypergraph has a vertex for each literal in A and each asserted literal; it has a directed hyperedge (i.e., a set of directed edges) for each asserting clause with more than one literal that is involved in the derivation. The hyperedge contains, for each of the false literals in the clause, one edge pointing to the the asserted literal.

Example 1 Consider the following CNF formula:

$$F = (a \vee b) \wedge (a \vee c) \wedge (a \vee d) \wedge (\neg b \vee e) \wedge (\neg c \vee \neg d \vee e).$$

Under partial assignment $\{\neg a\}$, literals b, c, and d are implied by the first three clauses of F and e is then implied by either the fourth or the fifth clause. The implication hypergraph for $F \cup \{\neg A\} \vdash e$ is shown in Fig. 2. Hyperedges are labeled with clause numbers. The number of edges in a hyperedge equals the number of literals in the corresponding clause minus one. Each node may be annotated with a *decision level* (the number following the @ sign in the figure). For the literals in the assignment, it is the order in which they are asserted (for instance, by a SAT solver). For an asserted literal, it is the highest level of its predecessors. Literals asserted by unit clauses have decision level equal to 0. The hypergraph of Fig. 2 shows that e can be implied in two different ways. □

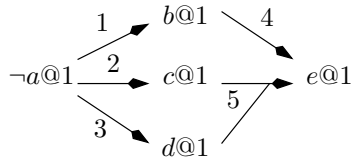

Figure 2: Implication hypergraph for Example 1.

SAT solvers usually keep track of just one way to assert a literal. Hence, they use an implication *graph* rather than a *hypergraph*.

Though DPLL only needs to be able to detect conflicting assignments to be complete, it is clearly advantageous for a DPLL-based SAT solver to work on a CNF formula that allows more to be done through deduction and less through enumeration. This motivates the following definition.

Definition 1 *For a given deductive system $\mathcal{S}$ and two equivalent sets of clauses F_1 and F_2, let A denote a partial assignment to the variables in $F_1 \cup F_2$. We say that F_1 has* deductive power *greater than or equal to F_2 (relative to $\mathcal{S}$) if and only if every A that is conflicting in F_2 is also conflicting in F_1 and for every A that is not conflicting in F_2 and any literal l such that $F_2 \cup A \vdash_{\mathcal{S}} l$, $F_1 \cup A \vdash_{\mathcal{S}} l$. (Note that if A is conflicting in F_1, then $F_1 \cup A \vdash_{\mathcal{S}} l$.)*

If F_1 has deductive power greater than or equal to F_2 (relative to $\mathcal{S}$) we write $F_2 \preceq_{\mathcal{S}} F_1$. If $F_2 \preceq_{\mathcal{S}} F_1$ and $F_1 \preceq_{\mathcal{S}} F_2$, then F_1 and F_2 have the same deductive power (relative to $\mathcal{S}$), written $F_1 \simeq_{\mathcal{S}} F_2$. If $F_2 \preceq_{\mathcal{S}} F_1$ and $F_1 \not\preceq_{\mathcal{S}} F_2$, we write $F_2 \prec_{\mathcal{S}} F_1$.

Since it is reflexive and transitive, $\preceq_{\mathcal{S}}$ is a preorder. In the following, unless otherwise stated, the deductive system is assumed to be $\mathcal{A}$ and we write $\vdash$ for $\vdash_{\mathcal{A}}$ and $\preceq$ for $\preceq_{\mathcal{A}}$. We are interested in transformations of a CNF formula that increase, or at least preserve, its deductive power. The following facts prove useful.

Lemma 1 *If $A = \{\neg l_1, \ldots, \neg l_{n-1}\}$ is a partial assignment to the variables of CNF formula F and $F \cup A \vdash l_n$, then $\{l_1, \ldots, l_n\}$ is a clause implied by F (that is, an* implicate *of F).*

PROOF. Suppose that a satisfying assignment for F included $\{\neg l_1, \ldots, \neg l_n\}$. Such an assignment would contradict $F \cup A \vdash l_n$. Therefore, any complete assignment that satisfies F must contain some literal l_i, $1 \leq i \leq n$. Hence, such assignment satisfies clause $\{l_1, \ldots, l_n\}$. □

Lemma 2 *Let F_1 be a CNF formula and let γ be an implicate of F_1. Let F_2 be the CNF formula obtained from F_1 by adding γ and optionally removing clauses that are subsumed by γ. Then, $F_1 \preceq F_2$.*

PROOF. F_1 and F_2 are obviously equivalent. Also, adding an implicate to a CNF formula cannot decrease its deductive power. For the removal of subsumed clauses, we need to consider two cases. Let A be an assignment and l be a literal such that $F_1 \cup A \vdash l$.

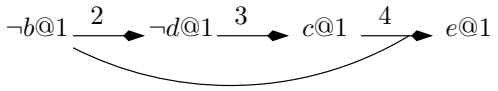

Figure 3: Implication hypergraph for Example 2.

Suppose γ' is a clause subsumed by γ that is used in the derivation of l. If γ' asserts a literal that is also in γ, then γ' can be replaced by γ in the derivation. If γ' asserts a literal not in γ, then all literals of γ are false in the derivation and adding γ to it leads to a conflict. In both cases, the conditions of Definition 1 are met. □

As a special case, since any clause of F is an implicate of F, one obtains the intuitive result that clauses subsumed by other clauses can be removed from a CNF formula without negatively affecting its deductive power. The next example shows how Lemmas 1 and 2 combine to improve a CNF formula that cannot be simplified by either subsumption or self subsumption.

Example 2 Consider the following CNF formula:

$$F = (a \vee b \vee c) \wedge (b \vee \neg d) \wedge (c \vee d) \wedge (b \vee \neg c \vee e).$$

Under partial assignment $\{\neg b\}$, $\neg d$ is implied by the second clause of F and c is implied by the third clause. The first clause is then satisfied. Finally, e is implied by the fourth clause. The implication hypergraph depicted in Fig. 3 shows that $(b \vee c)$ is an implicate of F that subsumes the first clause, and that $(b \vee e)$ is another implicate of F that subsumes the fourth clause. The simplified CNF is therefore

$$F' = (b \vee c) \wedge (b \vee \neg d) \wedge (c \vee d) \wedge (b \vee e).$$

Note that $F' \succ F$ because $F' \cup \{\neg e\} \vdash b$, but $F \cup \{\neg e\} \nvdash b$. In F' the first clause is the resolvent of the next two. Therefore,

$$F'' = (b \vee \neg d) \wedge (c \vee d) \wedge (b \vee e)$$

is equivalent to F'. One can verify that $F' \simeq F''$. □

Adding or removing a clause that is the resolvent of other clauses may or may not affect the deductive power of a CNF formula. Some clauses may never become asserting and therefore never appear in an implication hypergraph as shown in the next example.

Example 3 Consider the following CNF formula:

$$F = (\neg a \vee \neg b) \wedge (a \vee \neg d) \wedge (a \vee c) \wedge (\neg b \vee c \vee \neg d).$$

Assigning any literal of the last clause to false causes another literal of the same clause to be implied to true. Hence, that clause will never cause an implication and can be removed from F without affecting its deductive power.

Since $F \cup \{b\} \vdash \neg d$, $(\neg b \vee \neg d)$ is an implicate of F that subsumes $(\neg b \vee c \vee \neg d)$. In fact, $(\neg b \vee \neg d)$ is the resolvent of the first two clauses, but its addition does not improve deductive power. □

Even though adding an implicate to a CNF formula may not affect its deductive power, the situation is different when a conflict clauses containing a unique implication point (UIP) is learned by a DPLL-based solver. Such a clause is obtained from a clause that is conflicting under the current assignment by repeated resolution until only one literal (the UIP) has the same decision level as the last decision. After recalling a known result (Lemma 3, [9]), we show that the addition of a conflict learned clause containing a UIP always increases the deductive power.

Lemma 3 *Let F be a CNF formula and let γ be a conflict clause containing a UIP. Then γ is an implicate of F not subsumed by any clause of F.*

PROOF. Since γ is obtained by resolution of clauses in F, it is implied by the conjunction of those clauses, and hence by F. The conflict clause evaluates to false at the last decision level. Any clause that subsumes it should evaluate to false as well. However, all clauses that are false at the last decision level contain at least two literals that were assigned at the last decision level. (Otherwise they would have been asserting at some previous level.) On the other hand, γ contains exactly one literal assigned at the last decision level, namely the UIP. Therefore, it cannot be subsumed by any conflicting clause. □

Lemma 4 *Let F_1 be a CNF formula and let γ be a conflict clause containing a UIP. Let F_2 be the CNF formula obtained from F_1 by adding γ and optionally removing clauses that are subsumed by γ. Then, $F_1 \prec F_2$.*

PROOF. By Lemma 2, $F_1 \preceq F_2$. Let $\gamma = \{l_1, \ldots, l_n, u\}$ be the conflict clause and let u be its UIP. Consider the assignment $A = \{\neg l_1, \ldots, \neg l_n\}$. We have $F_2 \cup A \vdash u$, but $F_1 \cup A \nvdash u$, because otherwise u would have not had a higher decision level than the other literals. Hence, $F_2 \not\preceq F_1$. □

Lemmas 1 and 4 suggest a systematic approach to improving the deductive power of a CNF formula F. Suppose F contains no unit clauses. (If it does, simplify F in the obvious way.) Let $\gamma = \{l_1, \ldots, l_n\}$ be a clause of F. Consider the sequence of assignments $A_i = \{\neg l_1, \ldots, \neg l_i\}$ for $1 \leq i < n$ (the *assignment sequence* of γ). There exists a least i such that either $F \cup A_i$ is inconsistent, or $F \cup A_i \vdash l_j$, for $i < j \leq n$.

If A_i produces a conflict, we can add the learned conflict clause to F to increase its deductive power (thanks to Lemma 4). In any case, we use Lemma 1 to extract from the implication hypergraph an implicate of F that subsumes γ. This implicate may be γ itself, another clause of F that subsumes γ, or the resolvent of several clauses of F.

If γ is subsumed by another clause γ' already in F, then the implication hypergraph contains a hyperedge that asserts l_j and such that all the directed edges originate from literals in A_i. Therefore, even if finding a minimal implicate of F that subsumes γ is hard, removing from F a clause that is subsumed by another clause only requires inspecting the hypergraph induced by A_i. In Sect. 3 we shall see that keeping a single antecedent clause for each implied literal suffices. If a conflict occurs, and the implicate that contains l_j is not in subsumption relation with the conflict clause, then distillation of γ produces two clauses.

The "distillation" procedure outlined above may be used to detect some cases of so-called self subsumption. For instance, if $\gamma = \{l_1, l_2, l_3, l_4\}$ and $F = \{\gamma, \gamma'\}$, with $\gamma' = \{l_1, \neg l_2\}$, then $F \cup A_1 \vdash \neg l_2$. From that, it is possible to skip A_2 and conclude that $\{l_1, l_3, l_4\}$ is the desired implicate of F that subsumes γ.

Another example is given by $\gamma = \{l_1, l_2\}$ and $\gamma' = \{l_1, \neg l_2\}$. Asserting $\neg l_1$ leads to a conflict, and the learned clause, $\{l_1\}$, subsumes γ (and γ'). There is no guarantee, however, that self-subsumption will be detected. Consider $F = \{\gamma, \gamma'\}$, with $\gamma = \{l_1, l_2, l_3, l_4\}$ and $\gamma' = \{\neg l_1, l_2\}$. In this case, A_1 will cause no implications, and the simplified clause $\{l_2, l_3, l_4\}$ will not be discovered. Such limited ability should not surprise. In general, for a clause γ of F with n literals, n tries are sufficient to find an implicate of F that subsumes γ and cannot be further simplified by self subsumption. This is comparable to what the procedure of [4] does.

In the next section we detail a preprocessing algorithm that is based on *distilling* each clause γ of a CNF formula by trying its assignment sequence until either a conflict occurs or a literal l_j of

γ is asserted. Clause γ is replaced by either a conflict clause or an implicate containing l_j. The replacement for γ is guaranteed not to be subsumed by any other clause of F at the time it is generated. However, as distillation proceeds and shorter clauses are added to F, this property is lost. The process has to be iterated to convergence to guarantee that the resulting CNF formula is subsumption-free.

3. Algorithm Overview

In this section we describe an algorithm based on the distillation approach outlined in Sect. 2. The general idea is to test every clause of the given CNF formula with its sequence of assignments A_i. To gain efficiency, the clauses are initially stored in a *trie* [1] so that common prefixes may be identified. Distillation therefore consists of a depth-first traversal of the trie; at each node whose value is not yet asserted, a decision is made and its implications are deduced.

The deductions are computed by an implication procedure that extends the one used by standard DPLL. The objective of the extension is to be able to check implication hypergraphs for subsumption conditions. Hence, implication is continued until all pending assignments have been processed or a subsuming clause has been found. The antecedent of an asserted literal is allowed to change when another asserting clause for it is detected under appropriate conditions: Specifically, if the new antecedent is a clause that subsumes the current clause, or if the new antecedent has fewer literals than the current one. This latter provision is a heuristic that aims at identifying simple resolvents that subsume the tested clause. Only one antecedent is stored for each asserted literal, even though all are examined. This allows the checking for subsumption to be implemented with small modifications to a standard SAT solver.

Figure 4 shows the proposed algorithm. Procedure SIMPLIFYCNF() is given a set of clauses C as input, and returns a set of clauses D. From the input set of clauses C a trie is built. Variable $trie$ in SIMPLIFYCNF() is the set of roots of the trie. For each node in the trie, TRIEBASEDIMPLICATION() makes the decision on the children 'false' (0) and 'true' (1) only if they have siblings. BCPWITHSUBSUMPTION() detects a set of clauses that no existing clause in C subsumes, and adds them as output clauses to D. This check is based on the following result.

Lemma 5 *Let F be a set of clauses, Σ the SAT solver's assignment stack, $\gamma \in F$ a clause, and l a literal such that $F \cup \Sigma \vdash l$ and γ asserts l. Let $\gamma' \in F$ be a clause such that $\Sigma \wedge l \not\to \gamma'$. Such a clause γ' never subsumes γ.*

PROOF. Since $F \wedge \Sigma \rightarrow l$, $\gamma \subseteq \neg\Sigma \vee l$. If we assume that γ is subsumed by γ', then $\gamma' \subseteq \gamma \subseteq (\neg\Sigma \vee l)$. If $l \notin \gamma'$, the conflict should have already occurred before implying l in γ. If $l \in \gamma'$, since $\gamma' \setminus \{l\} \subseteq \neg\Sigma$, l should be implied in γ' too. Both cases are in contradiction with the assumption. Hence the clause γ is not subsumed by such a clause γ'. □

Thanks to Lemma 5 and the extended implication procedure, while testing a clause, it is possible to identify other clauses that are guaranteed not to be subsumed and move them to the output list directly.

In BCPWITHSUBSUMPTION(), the trie nodes associated with those implying clauses are set to mark in order to avoid for MARKTRIESOFIMPLYINGCLAUSES to test them in the future. When a conflict occurs in the implication procedure, TRIEBASEDIMPLICATIONAUX() generates a learned clause. If all the literals in the added conflict clause are decisions, the new clause subsumes the tested clause. Otherwise, we can replace the tested clause with another shorter conflict clause created by gathering decisions in the current assignment stack. TRIEBASEDIMPLICATION() may meet a literal that is already implied. In that case, RESOLVENTANALYSIS is called to search the implication graph for the decisions that are the source nodes to that literal. As in the case of a conflict, we can replace the clause currently tested with the clause generated by RESOLVENTANALYSIS.

```
1  SimplifyCNF(C) {
2     D = ∅;
3     Trie = BuildTrie(C);
4     TRIEBASEDIMPLICATION(Trie);
5     for each (c ∈ C) {
6        if (c.isMarkedClause)
7           D = D ∪ {c};
8     }
9     return D;
10 }

11 TRIEBASEDIMPLICATION(Trie) {
12    /* A is the assignment stack */
13    for each (t in Trie) {
14       if (t.isMarkedTrie)
15          continue ;
16       value = VALUE(t.node);
17       if (value != UNKNOWN) {
18          resolvent = RESOLVENTANALYSIS(
               traversedClause, t.node);
19          if (resolvent > 0)
20             SHORTENCLAUSEBYRESOLVENT(
                  traversedClause, resolvent);
21          if (t.child[value])
22             TRIEBASEDIMPLICATION(t.child[value]);
23          continue ;
24       }
25       TRIEBASEDIMPLICATIONAUX(t, 0);
26       TRIEBASEDIMPLICATIONAUX(t, 1);
27    }
28 }

29 TRIEBASEDIMPLICATIONAUX(t, value) {
30    if (t.child[value]) {
31       level = DECIDEBASEDONTRIE(t.node, value);
32       if (BCPWITHSUBSUMPTION(level) == Conflict) {
33          conflictClause = CONFLICTANALYSIS(level);
34          if (NUMOFLITERALS(conflictClause) < level)
35             ADDCONFLICTCLAUSE(conflictClause);
36          if (!ISEVERYLITERALBASEDONDECISION(
               conflictClause))
37             ADDCONFLICTCLAUSEBASEDONDECISIONS();
38       }
39       else {
40          MARKTRIESOFIMPLYINGCLAUSES(level);
41          TRIEBASEDIMPLICATION(t.child[value]);
42       }
43       BACKTRACK(level-1);
44    }
45 }
```

Figure 4: Algorithm overview

Table 1: Preprocessing Statistics.

CNF name	nVar	nCl	nLit	nOVar	nOCl	nOLit	nTrie	nImpA	nCnflA	nRes	mem	CPU
4blocks	759	47820	133940	641	24417	63954	31429	281515	659	2212	18.5	0.58
b-galileo-8	58075	294821	767187	38304	165042	360062	275245	7506395	682	516	91.0	8.55
b-galileo-9	63625	326999	852078	43156	304811	496364	316669	6736324	758	522	85.7	7.68
ibm-10	59057	323700	854093	28440	150801	355867	224633	107396019	75	2451	82.9	14.33
ibm-11	32110	150027	394770	21095	88578	213278	135689	1831694	324	2077	47.9	2.43
ibm-13	13216	65728	174164	10015	47601	119819	71831	1143194	183	452	29.4	1.32
bw-large.d	6326	131973	294118	4888	111315	249935	163722	2980629	417	153	43.2	3.74
rot	1452	2739	40341	1221	2283	9847	1939	868	10	6	10.1	0.03
hanoi4	719	4934	12200	456	4223	8399	3987	17122	47	8	10.4	0.03
c3540	5249	33199	112244	4696	32112	102293	62999	517649	1960	2082	20.1	1.54
c7552	11283	69529	234758	8341	63219	193521	126468	1287615	6224	5964	31.9	3.52
C880	1613	9373	31572	1539	9342	30376	18082	70336	389	377	12.4	0.2
dalu	9427	59991	202888	8196	58743	183248	110616	6804937	5194	5009	29.4	17.4
des	28903	179895	606506	23333	170599	551835	321934	15757473	6200	9665	65.9	40.0
frg1	3231	20575	69606	3226	20575	69528	36289	916238	27	26	15.2	2.35
frg2	10317	62943	211942	9510	62310	198487	115903	3304369	3602	3753	30.2	8.65
i8	12999	77941	262272	11822	73766	222255	225301	1517866	3516	4858	40.0	4.59
i10	14525	91139	307682	14213	90064	297083	168154	5809827	1436	3535	38.1	16.0
pre-s	12376	76080	260260	11351	73267	238265	144691	1498184	3427	4608	34.3	4.43
term1	3505	22229	75188	3218	21913	69917	41357	660725	1349	1528	15.8	1.7
am2910	26257	59389	138573	19991	58954	136808	93517	2015077	5	0	32.9	1.7
arbiter40	9157	12370	28862	4203	12341	28576	18977	121223	18	36	14.2	0.14
bakery20	27713	28683	66911	8556	26778	58712	44930	1354480	178	80	22.1	1.48
cube15	16015	27649	64493	8498	27344	59952	43831	1675370	131	29	20.4	1.26
daio100	115528	130046	303314	6683	60992	41407	179668	3073215	1523	477	63.7	4.03
daio150	181267	217021	506189	10356	98773	65165	286118	5193818	2471	575	98.9	7.35
eisenb19	27713	28683	66911	8556	26778	58712	44930	1354480	178	80	22.1	1.43
eisenb25	34301	45591	106355	14202	43669	98109	71420	1650982	177	89	29.7	1.9
bakery20	20483	21323	49739	6869	20821	47504	33098	472915	68	19	19.0	0.49
chame100	20923	47499	110775	15746	47028	109588	75265	533097	11	5	30.6	0.58
chame40	13675	29099	67863	9618	28725	66880	45969	332380	15	4	22.5	0.33
D15	44954	52897	123413	17414	52282	119920	83571	1460861	157	141	33.3	1.84
D16	47437	57920	135132	19102	57323	131654	91472	1515796	154	145	35.5	2.05
buffA	11504	21332	49748	6800	20386	47211	31969	1692907	23	11	17.8	1.27

4. Experimental Results

We have implemented the new distillation-based algorithm on top of the CirCUs SAT solver [6]. Table 1 shows the statistics of the preprocessor on a number of bounded model checking problems from [15], and verification and planning problems used in the SAT competitions [12]. The part of the table above the horizontal line is for the satisfiable instances, and the lower part is for the unsatisfiable instances. The experiments were run on IBM IntelliStation workstations with 1.7 GHz Pentium IV CPUs and 1 GB of RAM running Linux. For each SAT instance, the table gives the initial numbers of variables, clauses, and literals in Columns 2–4. The corresponding numbers for the output CNF are in Columns 5–7. The size of the trie built from the input CNF is shown in Column 8; it gives an idea of the reduction in assignments to be tried thanks to common prefixes. Columns 9–11 give some statistics on the running of the algorithm: the number of implications, the number of conflicts, and the number of resolution analyses for non-conflicting assignments. Finally, Columns 12–13 report memory consumption in MB and CPU time in seconds.

Table 2 reports the effects of preprocessing with Alembic and SatELite [4, 13] on the run times of CirCUs and MiniSat [5]. The variable elimination option of SatELite has been enabled at all of experiments. We have set time bound to 3600 s.

Alembic speeds up CirCUs for the unsatisfiable instances, as can be seen by comparing Columns 2 and 4. For the satisfiable instances, the total CPU time sometimes increases, though CirCUs itself is always faster with Alembic. This can be seen by comparing Table 2 to Table 1.

The results of CirCUs using SatELite are shown in the third column. Comparison of Columns 2 and 3 shows that the results are mixed. This is due, at least in part, to the fact that the current version of Alembic does not support variable elimination and self-subsumption, which are quite important for the bounded model checking experiments. Comparing Columns 5 and 6, one sees that for unsatisfiable instances, MiniSat benefits more from preprocessing with Alembic than from preprocessing with SatELite. Note that variable elimination is applied by MiniSat in both cases. For satisfiable instances, SatELite often produces the best CPU time. However, if the cost of simplification is excluded, the comparison between MiniSat with SatELite and MiniSat with Alembic shows that Alembic aids MiniSat to finish sooner in most instances.

In sum, a comparison of Alembic and SatELite reflects the different philosophies inspiring the two approaches. SatELite is usually faster, but is less effective for the hard unsatisfiable problems in which Alembic produces many conflicts.

Table 2: Performance of CirCUs and MiniSat 2.0 with different preprocessors. A+C: Alembic followed by CirCUs; S+C: SatELite followed by CirCUs; C: CirCUs with no preprocessing; S+M: SatELite followed by MiniSat. A+M: Alembic followed by MiniSat (without SatELite). T/O stands for more than 3600 s. An asterisk signals that the example is solved by Alembic.

CNF name	A+C	S+C	none	S+M	A+M
4blocks	0.79	0.57	0.87	0.53	0.88
b-galileo-8	10.05	7.87	6.56	3.46	10.05
b-galileo-9	12.48	7.03	6.92	4.7	9.38
ibm-10	26.01	14.73	28.45	5.75	17.8
ibm-11	14.34	10.12	39.7	3.4	4.43
ibm-13	18.17	2.88	58.41	8.29	6.36
bw-large.d	4.54	8.29	15.72	3.83	7.5
rot	0.04	0.04	0	0.04	0.04
hanoi4	11.21	20.42	13.23	0.11	0.25
c3540	1385.2	1536.2	1489.0	T/O	T/O
c7552	46.6	76.19	67.97	30.21	14.13
C880	23.09	25.21	34.51	275.04	195.3
dalu	T/O	T/O	T/O	T/O	T/O
des	305.92	328.84	339.66	180.98	124.52
frg1	T/O	T/O	T/O	T/O	T/O
frg2	386.15	469.36	485.31	541.52	805
i8	T/O	T/O	T/O	1713.5	1099.5
i10	T/O	T/O	T/O	T/O	T/O
pre-s	T/O	T/O	T/O	T/O	T/O
term1	360.15	672.03	497.48	T/O	1529.6
am2910	0.92	1.84	1.68	1.54	3.1
arbiter40	43.82	61.97	81.01	8.67	3.52
bakery20	13.37	1.81	20.87	1.83	2.0
cube15	326.26	162.53	471.52	79.96	44.42
daio100	4.03*	9.04	6.25	2.75	4.03*
daio150	7.35*	8.11	7.92	4.34	7.35*
eisenb19	13.24	1.84	20.93	1.83	1.95
eisenb25	107.78	13.87	158.88	14.7	25.6
bakery20	9.03	3.49	18.98	3.68	3.32
chame100	81.85	110.0	95	4.57	6.87
chame40	14.78	24.2	15.56	1.82	2.23
D15	690.88	94.97	755.1	24.79	22.16
D16	1944.4	391.81	2226	52.64	42.6
buffA	256.08	55.19	299.9	152.66	75

5. Conclusions

We have presented an efficient preprocessor for CNF satisfiability based on improving the deductive power of the given formula. Experimental results show that for some hard SAT instances, our preprocessor Alembic allows SAT solvers improved in performance. The techniques employed in Alembic have several other extensions and applications that we plan to explore, including: The extraction of small unsatisfiable cores; the application to restarts and solution enumeration; the extension to non-clausal reasoning; and logic synthesis and representation of sets by characteristic functions in CNF [10].

A more efficient implementation is also currently being developed to reduce the overhead for satisfiable instances and incorporate techniques such as variable elimination and self-subsumption.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, Reading, MA, 1983.

[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 Association for Computing Machinery*, 7(3):201–215, July 1960.

[4] N. Eén and A. Biere. Effective preprocessing in SAT through variable and clause elimination. In *Eighth International Conference on Theory and Applications of Satisfiability Testing (SAT 2005)*, St. Andrews, UK, June 2005. Springer-Verlag. LNCS 3569.

[5] N. Eén and N. Sörensson. An extensible SAT-solver. In *Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT 2003)*, pages 502–518, S. Margherita Ligure, Italy, May 2003. Springer-Verlag. LNCS 2919.

[6] 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)*, pages 519–522. Springer-Verlag, Berlin, July 2004. LNCS 3114.

[7] H. Jin and F. Somenzi. An incremental algorithm to check satisfiability for bounded model checking. *Electronic Notes in Theoretical Computer Science*, 2004. Second International Workshop on Bounded Model Checking. http://www.elsevier.nl/locate/entcs/.

[8] H. Jin and F. Somenzi. Strong conflict analysis for propositional satisfiability. In *Design, Automation and Test in Europe (DATE'06)*, pages 818–823, Munich, Germany, Mar. 2006.

[9] J. P. Marques-Silva and K. A. Sakallah. GRASP-a search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

[10] 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.

[11] 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.

[12] URL: http://www.lri.fr/∼simon/contest/results.

[13] URL: http://www.cs.chalmers.se/Cs/Research/ FormalMethods/MiniSat/MiniSat.html.

[14] N. Sörensson and N. Eén. MiniSat v1.13 – a SAT solver with conflict-clause minimization. In *Eighth International Conference on Theory and Applications of Satisfiability Testing (SAT 2005)*, St. Andrews, UK, June 2005. Springer-Verlag. LNCS 3569.

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

[16] L. Zhang. On subsumption removal and on-the-fly CNF simplification. In *Eighth International Conference on Theory and Applications of Satisfiability Testing (SAT 2005)*, St. Andrews, UK, June 2005. Springer-Verlag. LNCS 3569.

[17] 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, Nov. 2001.

[18] Q. Zhu, N. Kitchen, A. Kuehlmann, and A. Sangiovanni-Vincentelli. SAT sweeping with local observability don't cares. In *Proceedings of the Design Automation Conference*, pages 229–234, San Francisco, CA, July 2006.