Clause Simplification through Dominator Analysis*

Hyojung Han
University of Colorado at Boulder

Hoonsang Jin
Cadence Design Systems

Fabio Somenzi
University of Colorado at Boulder

Abstract

Satisfiability (SAT) solvers often benefit from clauses learned by the DPLL procedure, even though they are by definition redundant. In addition to those derived from conflicts, the clauses learned by dominator analysis during the deduction procedure tend to produce smaller implication graphs and sometimes increase the deductive power of the input CNF formula. We extend dominator analysis with an efficient self-subsumption check. We also show how the information collected by dominator analysis can be used to detect redundancies in the satisfied clauses and, more importantly, how it can be used to produce supplemental conflict clauses. We characterize these transformations in terms of deductive power and proof conciseness. Experiments show that the main advantage of dominator analysis and its extensions lies in improving proof conciseness.

1. Introduction

Satisfiability (SAT) solvers for propositional logic, in particular those based on the David-Putnam-Logemann-Loveland (DPLL) procedure [3, 2, 12, 13, 6], have found many applications in electronic design automation and artificial intelligence. Since the deduction procedure of a DPLL-based SAT solver is sound but not complete, its effects depend on which Conjunctive Normal Form (CNF) formula is chosen to represent the input function.

CNF transformations are among the most effective techniques to improve quality of the input formula by either simplifying clauses [4, 19, 21, 8, 9] or learning new ones [12]. In [8, 10], two notions that help in the design and evaluation of formula transformations have been introduced.[1] The first is *deductive power* of a CNF formula. It is motivated by the observation that the more consequences the DPLL procedure can deduce from each of its decisions, the more effective the pruning of the search space. The second notion is *proof conciseness*. It reflects the fact that the DPLL procedure progresses through the search space by proving that parts of that space contain no satisfying assignment and recording such findings in the form of new clauses.

These notions are at work in several techniques that are adopted by state-of-the-art SAT solvers to improve the quality of the CNF clauses. In PrecoSAT [1, 15], a clause with two literals may be derived based on *dominator* analysis during the deduction process. Such a clause tends to shorten the implication graph. In turn, a concise implication graph often benefits the recursive approach of [1] to minimize conflict clauses. In this paper, we discuss the effect of this type of clauses on deductive power and proof conciseness, and propose two main extensions of dominator-based analysis:

- A subsumption check concurrent with dominator computation, and
- the addition of dominator-based supplemental conflict clauses.

*This work was supported in part by SRC contract 2009-TJ-1859.

[1] A related notion of *empowering bi-asserting* clauses appeared in [14].

978-3-9810801-7-9/DATE11/ ©2011 EDAA

The rest of this paper is organized as follows. Background material is covered in Sect. 2. In Sect. 3 we describe the principles of dominator clauses. In Sect. 4 we show how to simplify asserting clauses during deduction. In Sect. 5 we discuss the use of dominator information in the simplification of satisfied clauses. In Sect. 6 we discuss dominator-based conflict clauses. Section 7 reports results from the implementation of the proposed approach. We draw conclusions and outline future work in Sect. 8.

2. Preliminaries

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. The CNF formula

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

therefore corresponds to the following propositional formula:

$$(\neg a \vee c)_1 \wedge (\neg b \vee c)_2 \wedge (\neg a \vee \neg c \vee d)_3 \wedge (\neg b \vee \neg c \vee \neg d)_4,$$

where subscripts identify clauses for ease of reference; c_i denotes the i-th clause of the formula. An *assignment* to the set of variables V of formula F is a mapping from V to $\{\mathsf{true}, \mathsf{false}\}$. A *partial* assignment maps a subset of V. A satisfying assignment for 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 $\{\{a\}, \{b\}, \{\neg d\}\}$. Given CNF formulae F_1 and F_2 over variable set 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 $F_2 \rightarrow F_1$. A clause γ is *asserting* under assignment A if all its literals except one (the asserted literal) are false. An asserting clause is an *antecedent* of its asserted literal.

Clause γ_1 *subsumes* clause γ_2 if $\gamma_1 \subseteq \gamma_2$. If $\gamma_1 \subseteq \gamma_2$, $\{\gamma_1\} \rightarrow \{\gamma_2\}$. Given $\gamma_1 = \gamma_1' \cup \{l\}$ and $\gamma_2 = \gamma_2' \cup \{\neg l\}$, the *resolution* of the two clauses over l produces the *resolvent* $\gamma_1' \cup \gamma_2'$, which is implied by $\{\gamma_1, \gamma_2\}$. Clauses γ_1 and γ_2 are in *self-subsumption* relation if their resolvent subsumes γ_1. For instance, $a \vee \neg b \vee c$ and $a \vee b$ give a resolvent $a \vee c$, which subsumes the former. If F contains clauses γ_1 and γ_2 such that γ_1 is in self-subsumption relation with γ_2, the CNF F' obtained by replacing γ_1 with the resolvent of γ_1 and γ_2 is equivalent to F.

We assume that the reader is familiar with the DPLL procedure [12], in which the solver maintains a current partial assignment that is extended until it either becomes a total satisfying assignment, or becomes conflicting. While extending the partial assignment, the solver uses deduction (also known as BCP: Boolean Constraint Propagation) to detect as many implications as possible by using asserting clauses. A derivation $F \cup A \vdash l$ (l is implied by CNF formula F together with partial assignment A) is conveniently rep-

resented by its *implication graph*. An implication graph has a vertex for each literal in A and each asserted literal l; it has a set of directed edges for each asserting clause with more than one literal that is involved in the derivation. Each vertex is annotated with a *decision level*, shown in the figure after the @ sign. The level gives the number of decisions that led to the assignment of the literal.

If extension of the assignment produces a conflict—that is, a clause, which is said to be *conflicting*, has all its literals assigned to false—the solver analyzes the conflict and *backtracks* accordingly [11]. Conflict analysis leads to learning a *conflict clause*, that is, a clause C with the following properties: given CNF formula F and assignment A, $F \rightarrow \{C\}$, $C \notin F$, and C is conflicting under A. A conflict clause is computed by resolving the conflicting clause with the antecedents of literals that appear in it. The antecedents are processed in reverse order in which the literals they assert were implied. Most conflict analysis algorithms terminate as soon as they find a clause containing a *Unique Implication Point* (UIP), that is, a single literal asserted at the current level. The conflict clause can be added to the given SAT instance to prevent the examination of regions of the search space that contain no solutions.

A SAT solver can simplify a conflict clause by dropping the literals implied at decision level 0 and those that are implied but other literals in the clause [5, 18]. This *conflict clause minimization* applies resolution to the conflict clause and the antecedent clauses.

Among the operations performed by a DPLL-based SAT solver, deduction plays a key role in boosting efficiency. Given a clause $\{l_1, \ldots, l_n\}$ and a partial assignment $\{\{\neg l_1\}, \ldots, \{\neg l_{n-1}\}\}$, the literal l_n is deduced. The deduction procedure in a DPLL-based SAT solver repeatedly applies this rule to asserting clauses in the given formula until either no new literal is implied, or a clause becomes conflicting. We denote this deduction procedure, which employs only this inference rule, by $\mathcal{D}$. We write $F \vdash_{\mathcal{D}} l$ if $\mathcal{D}$ deduces l from F. We write $F \vdash_{\mathcal{D}} \mathsf{false}$ if procedure $\mathcal{D}$ applied to F finds a conflicting clause. Procedure $\mathcal{D}$ is sound ($F \vdash_{\mathcal{D}} l$ implies $F \vdash l$ and $F \vdash_{\mathcal{D}} \mathsf{false}$ implies $F \vdash \mathsf{false}$) but not complete.

Since $\mathcal{D}$ is incomplete, its effects depend on which formula is used to describe the input function. In particular, the strengthening of clauses or the addition of new clauses may help. Though the DPLL procedure only needs to be able to detect conflicting assignments to be complete, it is clearly advantageous for a SAT solver based on it to work on a CNF formula that allows more to be done through deduction and less through enumeration. This motivated the introduction of the following definition and its use in the characterization of CNF formula transformations in [8, 10]. In it, it is assumed that when assignment A is conflicting in F, for every literal l, $F \cup A \vdash_{\mathcal{D}} l$.

Definition 1 *For given 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 $\mathcal{D}$) if and only if for every A and any literal l such that $F_2 \cup A \vdash_{\mathcal{D}} l$, $F_1 \cup A \vdash_{\mathcal{D}} l$. If F_1 has deductive power greater than or equal to F_2 (relative to $\mathcal{D}$) we write $F_2 \preceq_{\mathcal{D}} F_1$. If $F_2 \preceq_{\mathcal{D}} F_1$ and $F_1 \not\preceq_{\mathcal{D}} F_2$, we write $F_2 \prec_{\mathcal{D}} F_1$. If $F_2 \preceq_{\mathcal{D}} F_1$ and $F_1 \preceq_{\mathcal{D}} F_2$, then F_1 and F_2 have the same deductive power (relative to $\mathcal{D}$), written $F_1 \simeq_{\mathcal{D}} F_2$.*

In the following, we only consider deduction procedure $\mathcal{D}$ and write $\preceq$ for $\preceq_{\mathcal{D}}$ as well as $F_1 \simeq F_2$ for $F_1 \simeq_{\mathcal{D}} F_2$. In studying transformations of a CNF formula that increase, or at least preserve, its deductive power, the following facts are useful.

Lemma 1 ([10], Lemma 1) *Let F_1 be a CNF formula and let γ be an* implicate *of F_1 (that is, a clause implied by 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, $F_1 \preceq F_2$.*

Lemma 2 ([10], Lemma 8) *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 an implicate of F.*

Note that, as proved in [8], the addition of a conflict learned clause containing a UIP always increases the deductive power.

The clause $\{\neg l_0, l_n\}$, where $n \geq 2$, is a *transitive closure clause* of F, if there exist $l_1, \ldots, l_{n-1}$ such that $\{\neg l_0, l_1\}, \{\neg l_1, l_2\}, \ldots, \{\neg l_{n-1}, l_n\}$ are clauses of F.

Lemma 3 ([10], Lemma 2) *Let F be a CNF formula and $\gamma = \{\neg l_0, l_n\}$ be a transitive closure clause of F. Then $F \simeq F \cup \{\gamma\}$.*

While adding a transitive closure clause of the implications does not affect deductive power, it may shorten the implication graph. A more concise implication graph may benefit the procedures that work on it. For instance, the deduction procedure may identify a conflicting clause more quickly, and conflict analysis may resolve fewer antecedents. On the other hand, adding clauses to the CNF database indiscriminately may substantially slow down the deduction procedure. To prevent this, a supplemental clause should be generated only when its usefulness is established by an effective criterion.

3. Dominators

In this section we give an overview of the approach to learning new dominator-based clauses presented in [1] with the name of Lazy Hyper Binary Resolution (LHBR). The notion of *dominance* was introduced in [16] for the analysis of flow diagrams. This notion is readily adapted to implication graphs, as the following definition shows.

Definition 2 *Given an implication graph $G = (V, E)$, where V is the set of vertices and E is the set of directed edges, a node $d \in V$ dominates $v \in V$ if all paths from source nodes of G (decisions) to v go through d. Node d is the* earliest dominator *of v if it dominates v and has no other dominator than itself. Also, d* immediately dominates *v if it is the last dominator of v distinct from v. Finally, v is the* trivial *dominator of itself.* □

For each node v in G its *dominator set*, denoted by $\textsc{Dom}(v)$, contains every dominator of v. Under Definition 2, the dominators of a node are totally ordered and $v \in \textsc{Dom}(v)$.

A literal q is dominated by p in an implication graph for F if and only if $F \cup \{\{p\}\} \vdash_{\mathcal{D}} q$. Therefore, if q is dominated by p the clause $(\neg p \vee q)$ is an implicate of F by Lemma 2. We reserve the name *dominator clause* for the case in which $p \neq q$.

Example 1 Consider the following CNF formula F:

$$F = (\neg a \vee b)_1 \wedge (\neg b \vee c)_2 \wedge (\neg b \vee d)_3 \wedge (\neg b \vee \neg c \vee \neg d \vee e)_4 \ .$$

Suppose that the SAT solver makes decision $a@1$, which yields the implication graph shown in Fig. 1. In the implication graph, c, d, and e share two non-trivial dominators: the earliest dominator a and the immediate dominator b. Since $F \cup \{\{a\}\} \vdash_{\mathcal{D}} c$ and $F \cup \{\{a\}\} \vdash_{\mathcal{D}} d$, $\gamma_1 = (\neg a \vee c)$ and $\gamma_2 = (\neg a \vee d)$ are dominator clauses, and are transitive closure clauses. By Lemma 3, p adding γ_1 and γ_2 does not change the deductive power of F. On the other hand, $\gamma_3 = (\neg a \vee e)$ and $\gamma_4 = (\neg b \vee e)$ are dominator clauses that increase deductive power, since $F \cup \{\gamma_3\} \cup \{\neg e\} \vdash_{\mathcal{D}} \neg a$ but $F \cup$

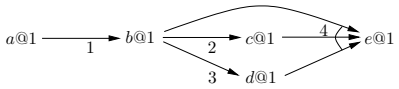


Figure 1: Implication graph of Example 1

$\{\neg e\} \not\vdash_{\mathcal{D}} \neg a$. (Similarly for γ_4.) In particular, since the asserting clause c_4 is subsumed by γ_4, it is removed if γ_4 is added. This leads to the shorter implication graph where the asserting clause c_4 is replaced with γ_4. Besides, $F \prec F \cup \{\gamma_3\} \prec F \cup \{\gamma_4\} \simeq F \cup \{\gamma_3, \gamma_4\}$. In this case, deducing the negation of the immediate dominator allows one to deduce the negation of all other non-trivial dominators. When the immediate dominator is distinct from the earliest dominator, the dominator clause involving the former is the one that usually gives the greatest boost to deductive power.

Dominator clauses do not always increase deductive power. If we consider

$$F' = F \wedge (e \vee \neg f)_5 \wedge (e \vee \neg g)_6 \wedge (f \vee g \vee \neg c)_7 \ ,$$

then γ_4 is still a dominator clause, but its addition does not affect deductive power. □

Example 1 motivates the CNF transformation implemented in [15] by adding dominator clauses, which are derived during the deduction procedure. Let c be a clause of CNF formula F. When literal l is deduced from c under a partial assignment, it is annotated with one of its dominators, d, which is then used to compute dominators for further implied literals.

The earliest dominator of l is easily computed. The earliest dominator of a decision is the literal itself. For an implied literal l, if all predecessors of l share the same earliest dominator d, then d is the dominator of l too; otherwise, l is the earliest dominator of itself.

In [15] a variation of this scheme is used. First, a dominator d is computed for vertex l with the above recursion in terms of the dominators chosen for the predecessors of l. These may not be earliest dominators; hence, d may not be the earliest dominator of l either. We call it the *recursive* dominator of l. Second, if the clause asserting l has two literals or d is trivial, l is annotated with d. Otherwise, the immediate dominator i of l is computed. If the negation of d appears in the asserting clause, then d is i. Otherwise, i is found by a search linear in the size of the subgraph of the implication graph between d and l. The search is made easy by enforcing the invariant that every predecessor of l is connected to d by exactly one path. (Alternatively, that the asserting clause of every literal l in the implication graph that is not its own dominator has exactly two literals: l and the negation of a dominator of l.)

When i is computed, l is annotated with it and the dominator clause $\gamma = (\neg i \vee l)$ is added to F. The implication graph is modified accordingly by making γ the antecedent of l. This simplifies the graph and guarantees that only one path connects i to l. If the negation of i is contained in the original antecedent clause c, c is subsumed by γ, as shown in Example 1. In the example, γ_4 computed from immediate dominator b simplifies c_4 while γ_3 based on the earliest dominator a does not.

The use of immediate dominators is motivated by the fact that, if there is a dominator clause that subsumes the asserting clause, then it contains the negation of the immediate dominator. PrecoSAT gives up the chance of finding some non-trivial dominators in return for the ability to simplify clauses using immediate dominators. Besides, Example 1 shows that dominator clauses including immediate dominators are also best for deductive power.

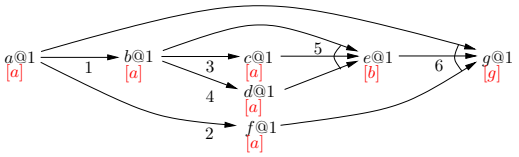


Figure 2: Implication graph of Example 2

Lemma 4 *If immediate dominator clauses are added for all implied literals with non-trivial dominators, then asserting $\neg l$ causes $\mathcal{D}$ to deduce the negation of all literals in* $\text{DOM}(l) \setminus \{l\}$.

4. Simplifying Clauses During Deduction

The use of immediate dominators increases the chances of subsumption of the asserting clause by the dominator clause. However, it may lead to missing non-trivial dominators.

Example 2 Given the following clauses:

$$(\neg a \vee b)_1 \wedge (\neg a \vee f)_2 \wedge (\neg b \vee c)_3 \wedge (\neg b \vee d)_4 \wedge$$
$$(\neg b \vee \neg c \vee \neg d \vee e)_5 \wedge (\neg a \vee \neg e \vee \neg f \vee g)_6 \ .$$

Suppose that $a@1$ is assigned as a decision. Propagating this assignment results in the implications shown in Fig. 2, where literals in square brackets are the dominators computed by the algorithm of [15]. Since b, c, d, and f are asserted by two-literal clauses, they are annotated with their earliest dominator a. For e asserted by c_5, its immediate dominator b is computed because c_5 does not contain the negation of the earliest dominator a. Literal b is used for the dominator computation when g is implied through c_6. However, since the other predecessor f has a different dominator from e, g is computed as its own dominator. This leads to missing the opportunity to simplify c_6 to $\gamma = (\neg a \vee g)$, which would be derived with the earliest dominator a. □

Example 2 shows that some simplification opportunities may be missed if vertices are labeled with their immediate dominators because fewer non-trivial dominators may be found. On the other hand, Example 1 shows that the exclusive use of earliest dominators may prevent other simplifications, when the immediate dominator is distinct from the earliest one. The simplifications of both approaches can be obtained within the same complexity bound by labeling each vertex with its earliest dominator, but computing the immediate dominator as well. Even the combined approach, however, misses some opportunities for simplification. Specifically, we can directly check for self-subsumption between the asserting clause and other implicates of F that may or may not be present in the database. We now show how this multistep resolution can be integrated with the search for the immediate dominator.

Example 3 Given the following clauses:

$$(\neg a \vee b)_1 \wedge (\neg b \vee c)_2 \wedge (\neg b \vee d)_3 \wedge (\neg c \vee e)_4 \wedge (\neg d \vee f)_5 \wedge$$
$$(\neg c \vee \neg d \vee \neg e \vee \neg f \vee g)_6 \ .$$

Suppose that propagating $a@1$ results in the implications shown in Fig. 3. Literal b is found as the immediate dominator of g in the graph. Since the asserting clause c_6 does not contain b, it cannot be simplified by the immediate dominator clause. However, c_6 can be simplified to $(\neg c \vee \neg d \vee g)$ because c and d imply e and f, respectively. □

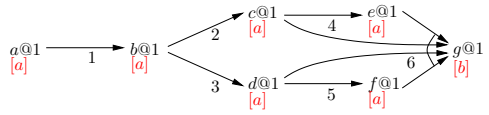


Figure 3: Implication graph of Example 3

```
1   ComputeDominatorAndSimplify(γ,rdom) {
2       idom = 0;
3       for each (antecedent literal lit in γ)
4           Ante(¬lit) = true;
5       for each (antecedent literal lit in γ)
6           idom = SearchDomAndSubsume(¬lit, rdom, idom);
7       for (lit = idom; lit != rdom; lit = Pred(lit))
8           Mark(lit) = false;
9       for each (antecedent literal lit in γ)
10          if (Ante(¬lit) == false)
11              Remove(γ, lit);
12          else Ante(¬lit) = false
13      return idom;
14  }

15  SearchDomAndSubsume(alit, rdom, idom) {
16      assert(rdom != alit);
17      lit = alit;
18      if (idom == 0) {
19          idom = alit;
20          do {
21              Mark(lit) = true;
22              lit = Pred(lit);
23              if (Ante(lit)) {
24                  Ante(idom) = false;
25                  for ( ;idom != lit; idom = Pred(idom))
26                      Mark(idom) = false ;
27              }
28          } while (lit != rdom);
29      } else {
30          while (!Mark(lit)) {
31              lit = Pred(lit);
32              if (lit == rdom) break ;
33              if (Ante(lit)) {
34                  Ante(alit) = false;
35                  return idom;
36              }
37          }
38          for ( ;idom != lit; idom = Pred(idom))
39              Mark(idom) = false ;
40      }
41      return idom;
42  }
```

Figure 4: Dominator analysis with simplifying asserting clauses

An antecedent literal a of a clause γ asserting l can be removed by self-subsumption with another implicate of F if each path between the immediate dominator of l and a goes through some other literal in γ. If only one path connects a literal to its dominator, the check is simple and efficient. Self-subsumption is possible even if the immediate dominator is not among the literals in γ.

The pseudocode for the check is shown in Figure 4. As in [15], the procedure is performed if the asserting clause γ has more than two literals and does not contain the negation of the recursive dominator. The immediate dominator is known to be on all paths connecting the recursive dominator "rdom" to the antecedent literals in γ. Therefore, it is known to be somewhere on the unique path between "rdom" and the first such literal, i.e., idom == 0 (line 18). All the vertices of the implication graph on that path are marked as candidate immediate dominators with Mark(lit) = true (lines 20–28). Tracing back (with Pred(l)) from the remaining literals, i.e., idom != 0 (line 29), until a marked literal is hit eliminates more candidates until the position of the immediate dominator is known (lines 30–39). Further, if a literal l in γ is found while tracing back from another literal a of γ, i.e., Ante(lit) == true (lines 23 and 33), then a is marked as redundant with Mark(idom) = false (line 24) and Mark(alit) = false (line 34). When this occurs during the initial path marking phase, l becomes the new endpoint of the marked path (lines 25–26). Otherwise, the trace back is terminated because the remaining work either was done or will be done when starting from l.

While this procedure is based on multistep resolution like on-the-fly simplification during conflict analysis [9] and conflict clause minimization [18], the use of the earliest dominator to limit the search makes it suitable for frequent use during deduction. The three procedures are complementary: dominator-based simplification resolves an asserting clause with clauses that precede it in the implication graph; on-the-fly simplification resolves an asserting clause with clauses that follow it in the implication graph, while conflict clause minimization does not modify clauses in the implication graph. Moreover, the simplification of the implication graph that results from replacing asserting clauses with dominator clauses speeds up the other two procedures.

5. Dominator Clauses and Redundancy

In this section we study when dominator clauses may duplicate existing clauses and how, on the other hand, dominator analysis may help a SAT solver remove redundant literals from clauses other than the asserting clauses and remove subsumed clauses from the database.

To minimize overhead, the SAT solver should not add dominator clauses that duplicate clauses already in F. While duplication is possible, if the SAT solver processes implications in the order in which it discovers them—which is the usual way—rather strong conditions must be met. These conditions for duplication are described in the following lemma.

Lemma 5 *Suppose implications are processed in first-in, first-out manner. Let $\gamma = \{\neg d, l\}$ be a dominator clause, where d is the dominator of l. If it is already present in formula F, γ is not an asserting clause in the implication graph and it subsumes the asserting clause for l.*

Proof. Let $\gamma = (\neg d \vee l)$ be a dominator clause computed for l from asserting clause c. Assume first that γ is asserting in the implication graph. Then c contains l and at least another literal l' that is deduced from d. Asserting d makes γ a unit clause so that l is implied through it before the implications of l' are examined. That prevents γ from being found as dominator clause, resulting in a contradiction. Suppose now γ is not in the implication graph, but is already in F. Suppose d is not in c. Then, l is implied from γ before it is implied from c. This prevents duplication. Therefore, d appears in c, and c is subsumed by γ. □

Lemma 5 suggests that duplication is not a frequent occurrence in solvers that preprocess their input and possibly remove more redundancies during DPLL. Besides, one can detect clauses subsumed by dominator clauses on-the-fly during the deduction procedure. That is, during the deduction procedure, if a clause c that is found to be satisfied by literal l contains the negation of dominator d of l, then c is subsumed by the dominator clause $(\neg d \vee l)$. The check for containment of the dominator of l may be expensive for

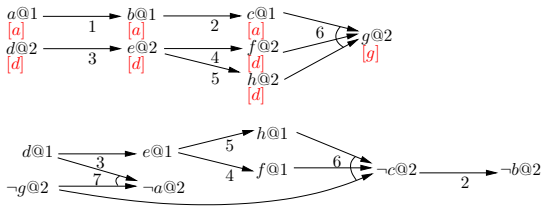


Figure 5: Implication graphs of Example 4

any dominator and for clauses with many literals. Hence, this approach should be applied with restraint: for example, only checking whether the recursive dominator of l (and possibly of a few more true literals in the clause) is the false literal that caused the clause to be examined. The annotation of each literal in the implication graph with its recursive dominator allows this test to be carried out even when the corresponding dominator clause is not added to the database. If not all satisfied literals are checked, subsumption may not be detected. On the other hand, subsumption may be found when the examined clause is subsumed by the dominator clause, even though it is not subsumed by the clause asserting l.

6. Dominator-Based Conflict Clauses

In this section we discuss the application of dominator analysis to the derivation of conflict clauses. Once a conflict clause is generated and possibly simplified, the dominator information collected for its antecedent literals can be used thanks to the following lemma.

Lemma 6 *Let $c = \{l_0, \ldots, l_n\}$ be the asserting clause of literal l_n. Let d_0 be a dominator of literal $\neg l_0$. Then, $\{\neg d_0\} \cup (c \setminus \{l_0\})$ is an implicate of F.*

Replacing antecedents with their non-trivial dominators therefore produces a new clause that can be used in conjunction with, or as replacement of, the clause computed by conflict analysis. Even though adding a dominator clause to a CNF formula may not affect its deductive power, when such a clause is derived from a conflict clause based on a UIP, it is at least guaranteed not to be a duplicate and it often improves deductive power.

Example 4 Suppose conflict clause $c_6 = (\neg c \vee \neg f \vee \neg h \vee g)$, where g is the UIP, is added and when after backtracking it becomes asserting, the implication graph is the one shown in Fig. 5 (upper). Suppose clause $c_7 = (\neg a \vee \neg d \vee g)$ is generated from c_6 by replacing the antecedent literals of g with their recursive dominators. Since c_7 contains a UIP (g), it can substitute c_6 as a conflict clause. When g is asserted by c_7 and the implication graph is shortened. To generate more compact implication graphs, we could always substitute a standard conflict clause like c_6 with a dominator-based conflict clause like c_7. However, this unlimited replacement may lead the SAT solver to miss some implications that it would have found with the standard conflict clauses. For instance, if c_7 replaces c_6, and later in the search, $d@1$ and $\neg g@2$ are assigned as decisions, literals $e@1$, $f@1$, $h@1$, and $\neg a@2$ are implied. However, if c_6 exists in the database, $\neg c$ and $\neg b$ are also implied as shown in the implication graph of Fig. 5 (lower). □

One may add both conflict clauses (e.g., c_6 and c_7 of Example 4), but the overhead is not negligible. Hence, supplemental conflict clauses based on dominators should be carefully generated and added to standard conflict clauses rather than replacing them.

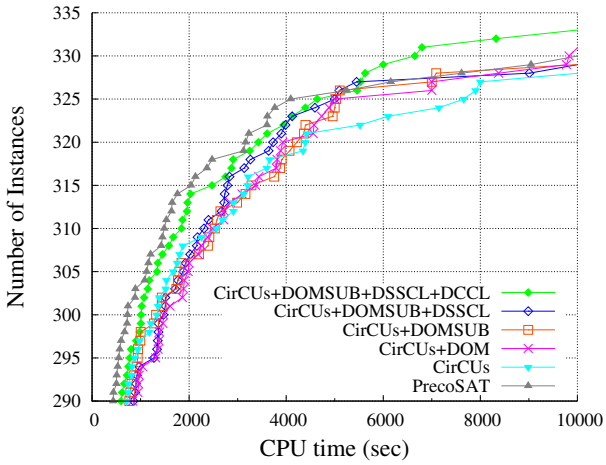


Figure 6: CPU time by PrecoSAT and CirCUs with and without proposed techniques

Our approach is to produce a dominator-based conflict clause γ' only when a newly-found conflict clause γ is obtained by on-the-fly simplification during conflict analysis [10]. In this case, since γ already exists in the database, adding γ' has an acceptable cost.

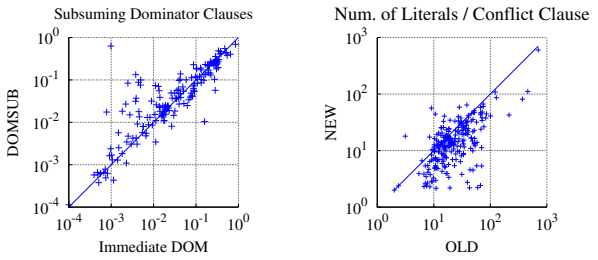
The replacement based on recursive dominators is straightforward and inexpensive: while computing the earliest dominator of the asserted literal, it is sufficient to check whether such dominator is the negation of one of the antecedent literals of the clause. This can be done by a single scan of the literals.

7. Experimental Results

We have implemented the algorithms for clause simplification during deduction and addition of dominator clauses in the CNF SAT solver *CirCUs* 2.0 [7, 20]. The benchmark suite is composed of all the CNF instances (with no duplicates) from the industrial category of the SAT Races of 2006, 2008, and the SAT Competitions of 2009 [17]. We conducted the experiments on a 2.4 GHz Intel Core2 Quad processor with 4GB of memory. We used 10000 seconds as timeout, and 2GB as memory bound. We tested PrecoSAT 236 [15] along with CirCUs 2.1 to provide a reference point. We denote the extensions by DOM (Dominator analysis), DOMSUB (Dominator analysis with Subsumption check on asserting clauses), DSSCL (Dominator-based Simplification on Satisfied Clauses), and DCCL (Dominator-based Conflict Clause generation).

The results are summarized in the graph of Fig. 6, which shows the number of instances completed in a given CPU time. From the graph it appears that the proposed techniques help CirCUs complete more instances within 10000 s, but provide limited benefits for simpler SAT problems. In fact, for the easier formulae, PrecoSAT is faster, but the improved CirCUs has performance close to that of PrecoSAT. Moreover, the proposed techniques tends to help CirCUs to solve more hard instances than the base version of CirCUs and PrecoSAT.

Figure 7 examine the effects of the proposed techniques (i.e., DOMSUB+DSSCL+DCCL) on (a) the number of dominator clauses subsuming asserting clauses per dominator computation and (b) the number of literals per conflict clause. For each of these quantities the geometric mean of the new/old ratios is reported (excluding cases in which one of the values is 0). Single-sample t-tests were performed to confirm the statistical significance of the data. The



(a) (b)

Figure 7: Effect of proposed techniques on (a)the number of subsuming dominator clauses per dominator computation: GEOMETRIC MEAN = 1.11, p-value = 0.001; (b)the number of literals per conflict clause: GEOMETRIC MEAN = 0.6, p-value = $2.2 \cdot 10^{-16}$

null hypothesis was that the geometric mean of the ratios is 1.

In Fig. 7(a), our dominator analysis, i.e., DOMSUB, produces more opportunities for computed dominator clauses to subsume asserting clauses than the analysis of immediate dominators. In our experiments, on average 0.02 literals were removed by subsuming dominator clauses for every implication.

Analysis of the CirCUs runs show that the major effect of DOMSUB+DSSCL+DCCL is in reducing the number of literals per conflict clause. (See Figure 7(b).) Our analysis indicates that this reduction stems from the reduction in the average number of resolution steps per conflict analysis. The reduction in resolution steps is 11% on average and the p-value is $1.05 \cdot 10^{-9}$. This translates in a 40% reduction in literals per conflict clause. In contrast, the indicators of increased deductive power are not changed in a decisive way. The number of conflicts per decision shows a 9% improvement on average and its p-value is 0.009. This supports the conclusion that the main way in which dominator clauses improve performance is by affecting proof conciseness.

8. Conclusions

Dominator analysis, introduced in PrecoSAT [1], is the basis for efficient techniques that allow a SAT solver based on DPLL to simplify the given CNF formula and learn new clauses while deducing new literals. In this paper, we have introduced two enhancements over the LHBR: a procedure to check clauses for simplification based on self-subsumption that is both more powerful and more efficient than analysis based on immediate dominators; and a low-overhead procedure to learn dominator-based conflict clauses.

We have shown that the CNF transformations based on dominator computation may improve the deductive power of the given formula, but primarily lead to more concise proofs. In our experiments, the new techniques were especially effective on large, difficult examples. We hope that a better understanding of the interplay between the new techniques and other components of the solver will lead to improved performance also for the easier SAT instances.

References

[1] A. Biere. P{re,i}coSAT@sc'09. SAT Competition 2009 - Solver Description, June 2009.

[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)*, pages 61–75, St. Andrews, UK, June 2005. Springer-Verlag. LNCS 3569.

[5] N. Eén, A. Mischchenko, and N. Sörensson. Applying logic synthesis for speeding up SAT. In *Theory and Applications of Satisfiability Testing: SAT 2007*, pages 272–286, Lisbon, Portugal, May 2007. Springer. LNCS 4501.

[6] 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.

[7] H. Han, H. Jin, H. Kim, and F. Somenzi. CirCUs 2.0 – SAT competition 2009 edition. SAT Competition 2009 - Solver Description, June 2009.

[8] H. Han and F. Somenzi. Alembic: An efficient algorithm for CNF preprocessing. In *Proceedings of the Design Automation Conference*, pages 582–587, San Diego, CA, June 2007.

[9] H. Han and F. Somenzi. On-the-fly clause improvement. In *Twelfth International Conference on Theory and Applications of Satisfiability Testing (SAT 2009)*, pages 209–222, Swansea, UK, June 2009. Springer-Verlag. LNCS 5584.

[10] H. Han, F. Somenzi, and H. Jin. Making deduction more effective in SAT solvers. *IEEE Transactions on COMPUTER-AIDED DESIGN of Integrated Circuits and Systems*, 29(8):1271–1284, 2010.

[11] J. P. Marques-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, Nov. 1996.

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

[13] 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.

[14] K. Pipatsrisawat and A. Darwiche. A new clause learning scheme for efficient unsatisfiability proofs. In *AAAI'08: Proceedings of the 23rd national conference on Artificial intelligence*, pages 1481–1484, Chicago, IL, 2008. AAAI Press.

[15] URL: http://fmv.jku.at/precosat.

[16] R. T. Prosser. Applications of boolean matrices to the analysis of flow diagrams. In *IRE-AIEE-ACM '59, eastern joint IRE-AIEE-ACM computer conference*, pages 133–138, New York, NY, Dec. 1959. ACM.

[17] URL: http://www.satcompetition.org.

[18] N. Sörensson and A. Biere. Minimizing learned clauses. In *Twelfth International Conference on Theory and Applications of Satisfiability Testing (SAT 2009)*, pages 237–243, Swansea, UK, June 2009. Springer-Verlag. LNCS 5584.

[19] N. Sörensson and N. Eén. MiniSat v1.13 – a SAT solver with conflict-clause minimization. SAT Competition 2005 - Solver Description, June 2005.

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

[21] 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.