Decision Procedures Should Be Able to Produce (Easily) Checkable Proofs

Allen Van Gelder
237 B.E., University of California, Santa Cruz, CA 95064
E-mail avg@cs.ucsc.edu.

July 15, 2002

Abstract

In many verification applications the desired outcome is that the formula is unsatisfiable: A satisfying assignment essentially exhibits a bug and unsatisfiability implies a lack of bugs, at least for the property being verified. Current high-performance satisfiability checkers and special-theory decision procedures are unable to provide proof of unsatisfiability. Since bugs have been discovered in many such programs long after being put into service, an uncheckable decision poses a significant problem if important economic or safety decisions are to be based upon it. Our thesis is that decision procedures can and should be designed to be able to output a proof.

1 Introduction

In recent research, planning problems, hardware and software verification problems and others have been encoded as satisfiability problems. There is a substantial difference among these types of problems, however. For planning problems, the successful outcome is a satisfying assignment, which describes the plan, and is easily checkable. For verification problems, the successful outcome is the *lack of* a satisfying assignment, which is not easily checkable.

A similar situation exists for many decision procedures for special theories: they are refutation procedures. For example, the Nelson-Oppen decision procedures for the quantifier-free theory of equality and the quantifier-free theory of LISP list structure, based on their congruence-closure procedure, negate the theorem, put that into disjunctive normal form (DNF), then check that each disjunct is unsatisfiable [NO80]. Many recent developments in the same vein have been reported [BGV01, VB01, BDL98, DD01].

Another active topic is combining decision procedures for different theories. The general idea is to transport constraints among underlying theories until the constraints in one of the underlying theories becomes unsatisfiable. Two influential early papers are by Nelson and Oppen [NO79] and Shostak [Sho84]. Some recent work is by Dill and co-authors [BDL96, BDS00]. Interestingly, Ruess and Shankar pointed out that many of these procedures contain essentially the same bug, and proposed a correction [RS01]. De Moura and Ruess [dMR02] describe a somewhat different approach, in which the underlying theories contribute constraints as propositional CNF and the unsatisfiability is checked in the latter theory, which is standard "SAT."

Our thesis is that decision procedures can and should be designed to be able to output a proof. While *finding* a proof is hard, *checking* a proof is straightforward. (By a "proof" we mean a real proof, with no steps omitted.) In practice, all underlying theories can produce a resolution proof. We shall argue that outputting a proof does not place an undue burden on the decision procedures.

2 Easily Checkable Proofs

If an important decision is to be taken based on claims that certain statements have been formally verified, there is a need to be able to verify the verifier. It is probably impractical to prove that the decision procedure is bug-free, but if it produces an easily checkable proof, then that *proof* can be verified without addressing the issue of whether the program is bug-free.

Propositional resolution proofs are very easy to check, independently of the program that produced the proof. The proof can be presented as a sequence of “records”. The m input clauses are indexed 1 through m in this sequence. After that, each record consists of its index in the sequence, the clashing literal, the two operand clauses (i.e., their indexes), and the resolvent clause. The correctness of the proof can be established merely by applying the definition of the resolution operation to each record in isolation. In theoretical terms, the checking problem is in logspace, a very easy complexity class.

First-order resolution proofs are only slightly more difficult to check. Besides the resolution operation, a new clause may be created through a substitution operation. Two clauses are resolvable only if they contain syntactically identical clashing literals. This method of presentation is simpler to check than the usual one, in which the substitution is embedded in the resolution operation. A substitution record consists of its index in the sequence, the single operand clause (i.e., its index), the substitution, and the resulting clause.

These considerations motivate our study of the problems connected with extracting proofs from the runs of SAT solvers and other decision procedures. In Section 5 we describe how to construct a refutation as a by-product of a DPLL search. In Section 6 we sketch how to construct a refutation as a by-product of congruence closure.

3 Notation

In CNF, the formula is a conjunction of clauses and each clause is a disjunction of literals; each literal is a propositional variable x or its negation $\neg x$. We denote a clause as $[q_1, q_2, \ldots, q_k]$ and a formula as $\{C_1, C_2, \ldots, C_m\}$. An empty formula is *true* and [], the empty clause, is *false*. We also define the *tautologous clause* $\top$, which is true under any assignment.

Definition 3.1: (strengthened formula) Let $\mathcal{A}$ be a partial assignment for formula $\mathcal{F}$. The clause $C|\mathcal{A}$, read “C strengthened by $\mathcal{A}$”, and the formula $\mathcal{F}|\mathcal{A}$, read “$\mathcal{F}$ strengthened by $\mathcal{A}$”, are defined as follows.

1. $C|\mathcal{A} = \top$, if C contains any literal that occurs in $\mathcal{A}$.
2. $C|\mathcal{A} = C - \{q \mid q \in C \text{ and } \neg q \in \mathcal{A}\}$, if C does not contain any literal that occurs in $\mathcal{A}$. This might be the empty clause.
3. $\mathcal{F}|\mathcal{A} = \{C|\mathcal{A} \mid C \in \mathcal{F}\}$; i.e., apply strengthening to each clause in $\mathcal{F}$.

   Usually, occurrences of $\top$ (produced by part (1)) are deleted in $\mathcal{F}|\mathcal{A}$.

The operation $\mathcal{F}|p$ (i.e., $\mathcal{F}|\{[p]\}$) is sometimes called “unit simplification”. □

The resolution operation is denoted by res(q, C_0, C_1); it returns the resolvent of clauses C_0 and C_1 with clashing literal q. The resolvent is $(C_0 - [q]) \cup (C_1 - [\neg q])$.

4 SAT Solvers

Nearly all complete satisfiability solvers are in the DPLL family (for Davis, Putnam, Loveland, and Logemann [DLL62]). They search for a satisfying assignment by fixing variables one by one and backtracking when an assignment forces the formula to be *false*. The procedure is not very effective in its original form, but it has been enhanced with various techniques to reduce the search space. Reasoning techniques can be broadly classified as *preorder* and *postorder*.

Preorder techniques are applied as the search goes forward, and include binary-clause reasoning, equivalent-literal identification, and other efficient reasoning steps whose goal is to show that certain variable bindings cannot lead to a satisfying assignment [BS92, Pre95, VGT96, Li00, Bac02]. (The omission of the unit-clause rule is intentional, as explained in Section 5.)

Postorder techniques are applied when the search is about to backtrack, because a "conflict" has been discovered [SS96, Zha97, BS97, MMZ$^+$01]. Postorder techniques are variously called non-chronological backtracking, conflict-directed back-jumping, and learning. These techniques are compared in a recent paper [ZMMM01].

There are substantial difficulties in combining preorder and postorder techniques; Van Gelder has reported one prototype [VG02a], and Bacchus uses both reasoning styles to some extent [Bac02].

5 DPLL as Construction of a Refutation

We briefly review how preorder and postorder techniques can be combined in a SAT solver based on DPLL, based on [VG02a]. The difficulty to be overcome is that derived clauses depend on the assumptions (guessed assignments) in a nontransparent way. If the only reasoning is unit resolution, then each derived clause corresponds to one original clause, and the relevant assumptions can be traced through this association [LP92, SS96, VGO99]. With non-unit resolution, a derived clause may be associated with an arbitrary number of original clauses. Our program, called "`2cl`", constructs a directed acyclic graph (DAG) to maintain the association. This DAG maintains the information necessary to output a resolution refutation for an unsatisfiable formula.

A key to understanding the correctness of the procedure is that the DPLL algorithm can be viewed dually as a procedure to construct a resolution refutation. The refutation is constructed in a post-order fashion. Conflict-directed back-jumping (CBJ) falls out naturally from this dual view. This view is then enhanced with resolutions performed during the search.

Implementation of the method in C posed challenges in memory management. The auxiliary data structure for the directed acyclic graph can be built with only a constant amount of overhead per operation, but during backtracking large amounts of the structure become garbage.

5.1 DPLL without Preorder Reasoning

Normally, the classical branching algorithm of Davis, Putnam, Loveland, and Logemann (DPLL) [DP60, DLL62] is viewed as a backtracking search for a satisfying assignment for a Boolean CNF formula. It can be sketched as a recursive procedure with parameters $\mathcal{F}$ and $\mathcal{A}$, the formula and the assumptions (guessed assignments, represented as a set of literals):

DPLL($\mathcal{F}$, $\mathcal{A}$)

If $\mathcal{F}|\mathcal{A}$ has no clauses:
    output "sat by $\mathcal{A}$" and **terminate**.
If $\mathcal{F}|\mathcal{A}$ contains an empty clause:
    return "unsat".

(Otherwise) Choose a splitting literal q.
Call DPLL($\mathcal{F}$, $\{\mathcal{A}, \neg q\}$).
Call DPLL($\mathcal{F}$, $\{\mathcal{A}, q\}$).
Return "unsat".

The top-level call is DPLL($\mathcal{F}$, $\emptyset$). For implementation efficiency, the first parameter is normally $\mathcal{F}|\mathcal{A}$, rather than $\mathcal{F}$. Two important observations are:

1. *Unit-Clause Rule*: Note that the unit-clause rule is incorporated in the above sketch by choosing q to be a unit clause if $\mathcal{F}|\mathcal{A}$ contains any such.

2. *Pure-Literal Rule*: Note that the pure-literal rule is incorporated in the above sketch by *not* choosing q if it is a pure literal (unless all remaining variables are pure literals).

Thus the procedure incorporates all of DPLL.

If the formula is satisfiable, the pseudocode outputs and terminates, rather than returning back out of the recursion. This is not recommended for actual implementation, but it simplifies the presentation to focus on the processing of unsatisfiable formulas.

A dual view of this procedure is that it is constructing a resolution refutation. The procedure returns a *resolution tree* whose *root* contains the clause derived by the tree and whose *leaves* are clauses in $\mathcal{F}$. Resolution trees are denoted by P_0 and P_1. Each internal node contains the resolvent clause of its two children.

Refute($\mathcal{F}$, $\mathcal{A}$)

If $\mathcal{F}|\mathcal{A}$ has no clauses:
    output "sat by $\mathcal{A}$" and **terminate**.
If $\mathcal{F}|\mathcal{A}$ contains an empty clause:
    return a clause of $\mathcal{F}$ that became empty.

(Otherwise) Choose a splitting literal q.
P_0 = Refute($\mathcal{F}$, $\{\mathcal{A}, \neg q\}$).
P_1 = Refute($\mathcal{F}$, $\{\mathcal{A}, q\}$).
Return the tree for res(q, *root*(P_0), *root*(P_1)).

We confine this discussion to $\mathcal{F}$ being unsatisfiable. The objective of Refute($\mathcal{F}$, $\mathcal{A}$) is to return the derivation of a clause C such that $\neg C \subseteq \mathcal{A}$. (We are being loose about notation, regarding $\neg C$ as a set of literals and using $\{\mathcal{A}, q\}$ to denote the addition of q to the set $\mathcal{A}$.) If Refute($\mathcal{F}$, $\mathcal{A}$) always achieves its objective, then, since $\mathcal{A}$ is empty in the top level call, the value returned to top level is a derivation of the empty clause.

It is clear that Refute($\mathcal{F}$, $\mathcal{A}$) *does* achieve its objective in the nonrecursive case, where it encounters an empty clause. Otherwise, if both recursive calls meet *their* objectives, then $\neg$*root*$(P_0) \subseteq \{\mathcal{A}, \neg q\}$ and $\neg$*root*$(P_1) \subseteq \{\mathcal{A}, q\}$. By the definition of resolution, $\neg$ res(q, *root*(P_0), *root*(P_1)) $\subseteq \mathcal{A}$.

The only gap in the above argument is that possibly *root*(P_0) does not contain q, so that resolution with q as the clashing literal is not defined. But then $\neg$*root*$(P_1) \subseteq \mathcal{A}$, so Refute($\mathcal{F}$, $\mathcal{A}$) can simply return P_0 and meet its objective. Similarly, if *root*(P_1) does not contain $\neg q$, then Refute($\mathcal{F}$, $\mathcal{A}$) can return P_1. With these added details the algorithm is correct. Figure 1 illustrates the ideas.

Now we observe that if *root*(P_0) does not contain q, then the call Refute($\mathcal{F}$, $\{\mathcal{A}, q\}$) is unnecessary. The right half of the refutation tree can be pruned and the left half becomes the whole tree. Thus conflict-directed back-jumping is built into this algorithm!

In other words, $\neg$*root*(P_0) is a conflict set for $\mathcal{F}|\,\{\mathcal{A}, \neg q\}$. If *root*($P_0$) does not contain q, then $\neg$*root*(P_0) is also a conflict set for $\mathcal{F}|\mathcal{A}$, and the assumption $\neg q$ was irrelevant to the former formula being unsatisfiable.

5.2 Incorporating Resolution on the Way Down

We now consider an enhanced version of Refute($\mathcal{F}$, $\mathcal{A}$) in which some resolution steps may be carried out prior to the recursive call. The new procedure is Refute($\mathcal{F}$, $\mathcal{D}$, $\mathcal{A}$), where $\mathcal{D}$ denotes a set of derived clauses. Also, Δ will denote a set of clauses derived after Refute($\mathcal{F}$, $\mathcal{D}$, $\mathcal{A}$) begins.

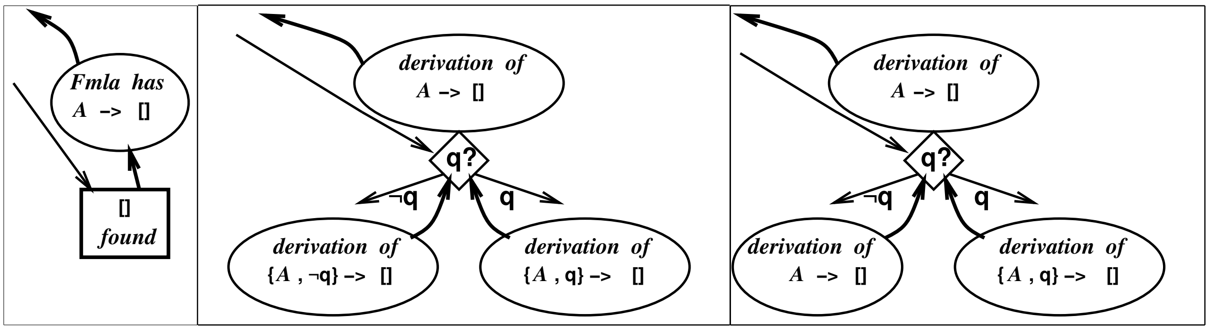

Figure 1: Extracting a resolution refutation from DPLL (as Refute). Note that the implication $\mathcal{A} \to []$ is equivalent to a disjunctive clause. In general, the antecedents are *subsets* of A, $\{A, \neg q\}$, and $\{A, q\}$, rather than the entire sets. Left panel is the base case; middle panel shows resolution, the usual case; right panel applies when a missing clashing literal prevents resolution.

Refute($\mathcal{F}$, $\mathcal{D}$, $\mathcal{A}$)

If $\mathcal{F}|\mathcal{A}$ has no clauses:
    output "sat by $\mathcal{A}$" and **terminate**.
If $\mathcal{F}|\mathcal{A}$ contains an empty clause:
    return a clause of $\mathcal{F}$ that became empty.
If $\mathcal{D}|\mathcal{A}$ contains an empty clause:
    return a proof tree for a clause of $\mathcal{D}$ that became empty.
(Otherwise) Derive additional clauses Δ.
If $\Delta|\mathcal{A}$ contains an empty clause:
    return a proof for a clause of Δ that became empty.

(Otherwise) Choose a splitting literal q.
P_0 = Refute($\mathcal{F}$, $\{\mathcal{D}, \Delta\}$, $\{\mathcal{A}, \neg q\}$).
If q is not in *root*(P_0):
    return P_0.
(Otherwise) P_1 = Refute($\mathcal{F}$, $\{\mathcal{D}, \Delta\}$, $\{\mathcal{A}, q\}$).
If $\neg q$ is not in *root*(P_1):
    return P_1.
(Otherwise) Return the tree for res(q, *root*(P_0), *root*(P_1)).

As long as each clause in $\mathcal{D}$ or Δ is derived by resolution, its derivation can be represented by a resolution DAG (directed acyclic graph) whose edges point to earlier-derived clauses or original clauses. Such DAGs are returned in the nonrecursive case of Refute($\mathcal{F}$, $\mathcal{D}$, $\mathcal{A}$). Thus the structure returned is still a valid resolution proof, but it is not necessarily a tree.

5.3 Finding Conflict Sets in Practice

Section 5.2 gives us the theoretical basis for constructing a resolution refutation of a given formula $\mathcal{F}$. Each derived clause that is returned by Refute($\mathcal{F}$, $\mathcal{D}$, $\mathcal{A}$) is a conflict set for $\{\mathcal{F}, \mathcal{D}\}\,|\mathcal{A}$. These conflict sets can be used to prune unnecessary backtracking. A conflict set $\neg P$ can be represented very nicely as a descending-order list of the depths at which the literals of $\neg P$ were assumed or guessed. Once conflict sets are materialized, they can be combined in time that is linear in their combined length.

The main problem is materializing the conflict set for the resolution DAG returned in the nonrecursive case. Notice that this is a relatively simple matter for Refute($\mathcal{F}$, $\mathcal{A}$) because the returned value is always the trivial DAG consisting of a clause of $\mathcal{F}$, say C, such that $\neg C \subseteq \mathcal{A}$. For Refute($\mathcal{F}$, $\mathcal{D}$, $\mathcal{A}$) it would not be practical to materialize every derived

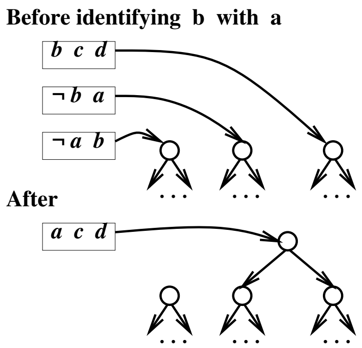

Figure 2: Resolution-DAG nodes need to be separate from their clauses because the clauses can disappear.

clause in case it happened to be useful. Instead, if D is a conceptually derived clause, only $D|\mathcal{A}$ is materialized and two pointers are maintained to the ancestor clauses that were resolved to produce $D|\mathcal{A}$.

Following pointers from a certain clause leads to the assumed literals that were used to derive that clause. The idea is illustrated in Figure 2.

Suppose $D|\mathcal{A}$ is the empty clause. To reconstruct D it suffices to traverse the DAG rooted at $D|\mathcal{A}$ and collect all the reachable assumptions. This is accomplished efficiently with depth-first search. (Efficiently does not mean inexpensively.) The justification is that every literal that occurs in a clause (not an assumption) that participated in the derivation of D either survives in D or was resolved away during the derivation. If $\neg q$ is in $\mathcal{A}$ and is reached in the DAG, then q must be in an ancestor-clause of D and also in D. In fact, the ancestors collected comprise a conflict set for $\{\mathcal{F}, \mathcal{D}\}\,|\mathcal{A}$.

5.4 Equivalent-Literal Processing

Equivalent-literal identification can also be justified in the proof by resolution. If $a = b$ has been derived, and a is chosen as the leader, use $[\neg b, a]$ to resolve out b and use $[b, \neg a]$ to resolve out $\neg b$. Implementation issues are discussed elsewhere [VG02b]. Two strategies are (1) to resolve out b and $\neg b$ throughout the formula as soon as $a = b$ is derived, and (2) to record the information in a disjoint-sets data structure and wait until the clause is needed for the proof. Both approaches have the potential to incur sizable overhead *whether or not the procedure outputs a proof.*

6 Extracting a Proof from Congruence Closure

The Nelson-Oppen congruence-closure procedure can be viewed as a *strategy* for choosing proof operations. We will sketch the main idea for the quantifier-free theory of equality with uninterpreted functions. For simplicity, let us assume that the formula whose inconsistency is to be established contains a unary function f and a binary function g and is in disjunctive normal form (DNF). Actually, a separate resolution refutation is constructed for each conjunction in the DNF formula.

The notation τ_i represents some term made from these function symbols, variables and constants, which occurs in the formula. We have the first-order equality axioms in the background (i.e., when we talk about terms and atoms in the formula, we exclude these):

$$x = x \tag{1}$$

$$y \neq x \quad \vee \quad x = y \tag{2}$$

$$x \neq y \vee y \neq z \quad \vee \quad x = z \tag{3}$$

$$v \neq x \quad \vee \quad f(v) = f(x) \tag{4}$$

$$v \neq x \vee w \neq y \quad \vee \quad g(v, w) = g(x, y) \tag{5}$$

For each conjunction of equality literals (positive and negative) A dynamic equivalence relation ($\equiv$) is constructed by starting with the positive equalities: if $\tau_1 = \tau_2$ is a positive literal, set $\tau_1 \equiv \tau_2$. For any positive atom $f(\tau_1) = f(\tau_2)$ in the conjunction such that $\tau_1 \equiv \tau_2$, make $f(\tau_1) \equiv f(\tau_2)$ *and output the following proof steps*:

1. Let τ_3 be the leader of the equivalence class containing τ_1 and τ_2. Apply the substitution $x \leftarrow \tau_1$, $y \leftarrow \tau_3$, $z \leftarrow \tau_2$ to Eq. 3, creating

$$\tau_1 \neq \tau_3 \vee \tau_3 \neq \tau_2 \quad \vee \quad \tau_1 = \tau_2. \tag{6}$$

2. Resolve Eq. 6 with unit clauses $\tau_1 = \tau_3$ and $\tau_3 = \tau_2$, which are either in the input formula or were derived earlier as part of a congruence operation. This outputs the unit clause

$$\tau_1 = \tau_2. \tag{7}$$

3. Apply the substitution $v \leftarrow \tau_1$, $x \leftarrow \tau_2$ to Eq. 4, creating

$$\tau_1 \neq \tau_2 \quad \vee \quad f(\tau_1) = f(\tau_2). \tag{8}$$

4. Resolve Eq. 8 and Eq. 7 to create the unit clause

$$f(\tau_1) = f(\tau_2). \tag{9}$$

5. Apply the substitution $x \leftarrow f(\tau_2)$, $y \leftarrow f(\tau_1)$ to Eq. 2, creating

$$f(\tau_1) \neq f(\tau_2) \quad \vee \quad f(\tau_2) = f(\tau_1). \tag{10}$$

6. Resolve Eq. 10 and Eq. 9 to create the unit clause

$$f(\tau_2) = f(\tau_1). \tag{11}$$

For any positive atom $g(\tau_1, \tau_3) = g(\tau_2, \tau_4)$ in the conjunction such that $\tau_1 \equiv \tau_2$ and $\tau_3 \equiv \tau_4$, make $g(\tau_1, \tau_3) \equiv g(\tau_2, \tau_4)$ *and output proof steps similar to those described above for f.*

Repeat these congruence operations until no new congruences can be found. Then, if any negative atom $\tau_1 \neq \tau_2$ is found in the conjunction, such that $\tau_1 \equiv \tau_2$, simply resolve it with $\tau_1 = \tau_2$, which must have been derived during congruence closure on this conjunction, to produce the empty clause. If no such literal is found, the formula is satisfiable, as described in the original paper [NO80].

A point we glossed over is that the clause $\tau_1 = \tau_3$ might not have been output as soon as τ_3 became the leader of the equivalence class for τ_1. However, the first time `find(`τ_1`)` is called, as a by-product of path compression, this clause can be derived.

Notice that the overhead of creating the proof only increases the constant factor for the entire decision procedure. It is completely mechanical. The technique is not limited to a strategy that is based on DNF. It can be adapted to any strategy for application of the equality axioms.

7 Conclusion

We have argued that decision procedures can be designed to output proofs that support their decision and are easily checkable by an entirely independent program. We believe that this discipline can assist in debugging the decision procedures as they are developed. In languages that support conditional compilation, the code to output the proof can be encapsulated, so that the program can be compiled to skip this output in the interest of running faster. When an important decision needs to be verified, the program can be rerun with proof-output enabled.

Another implementation consideration is that parts of the proof output might not contribute to the derivation of the empty clause. A post-processing step might be useful to discard unreferenced clauses and renumber the useful clauses to eliminate gaps. This makes it easier for the proof checker to keep the clauses in an array for fast lookup. To keep the overhead of proof-output down, both in time and space, the initial output should be in binary. With multiple output buffers, the delays of disk latency can be largely avoided. We recognize that technical implementation issues like these are not of interest to most researchers, but they are important to make the move from research to production.

Finally, we advocate the establishment of competitions for verification procedures that are able to produce proofs. The CADE conference has competitions for theorem provers that are able to produce proofs, as well as those that only produce decisions. However, their emphasis is on first-order provers that do not have a large propositional component. The current propositional competitions offer no incentive to develop procedures that are able to produce proofs.

References

[Bac02] F. Bacchus. Exploring the computational tradeoff of more reasoning and less searching. In *Symposium on the Theory and Applications of Satisfiability Testing*, pages 7–16, Cincinnati, OH, 2002.

[BDL96] Clark Barrett, David Dill, and Jeremy Levitt. Validity checking for combinations of theories with equality. In *Formal Methods In Computer-Aided Design*, volume 1166 of *LNCS*, pages 187–201, Palo Alto, CA, 1996. Springer-Verlag.

[BDL98] Clark W. Barrett, David L. Dill, and Jeremy R. Levitt. A decision procedure for bit-vector arithmetic. In *35th Design Automation Conference*, San Francisco, 1998.

[BDS00] Clark W. Barrett, David L. Dill, and Aaron Stump. A framework for cooperating decision procedures. In *17th International Conference on Computer-Aided Deduction*, 2000.

[BGV01] R. E. Bryant, S. German, and M. N. Velev. Processor verification using efficient reductions of the logic of uninterpreted functions to propositional logic. *ACM Transactions on Computational Logic*, 2(1), 2001.

[BS92] A. Billionnet and A. Sutter. An efficient algorithm for the 3-satisfiability problem. *Operations Research Letters*, 12:29–36, July 1992.

[BS97] R. J. Bayardo, Jr. and R. C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings Fourteenth National Conference on Artificial Intelligence (AAAI-97)*, pages 203–208, 1997.

[DD01] Satyaki Das and David L. Dill. Successive approximation of abstract transition relations. In *IEEE Symposium on Logic in Computer Science*, Boston, 2001.

[DLL62] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5:394–397, 1962.

[dMR02] L. de Moura and H. Ruess. Lemmas on demand for satisfiability solvers. In *Symposium on the Theory and Applications of Satisfiability Testing*, pages 244–251, Cincinnati, OH, 2002.

[DP60] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the Association for Computing Machinery*, 7:201–215, 1960.

[Li00] C. M. Li. Integrating equivalency reasoning into davis-putnam procedure. In *AAAI*, 2000.

[LP92] S.-J. Lee and D. A. Plaisted. Eliminating duplication with the hyper-linking strategy. *Journal of Automated Reasoning*, 9(1):25–42, August 1992.

[MMZ+01] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *39th Design Automation Conference*, June 2001.

[NO79] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, 1979.

[NO80] G. Nelson and D. C. Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM*, 27(2):356–364, 1980.

[Pre95] D. Pretolani. Efficiency and stability of hypergraph SAT algorithms. In D. S. Johnson and M. Trick, editors, *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge.*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society, 1995.

[RS01] H. Ruess and N. Shankar. Deconstructing shostak. In *IEEE Symposium on Logic in Computer Science*, Boston, 2001.

[Sho84] R. E. Shostak. Deciding combinations of theories. *Journal of the ACM*, 31(1):1–12, 1984.

[SS96] J. P. Silva and K. A. Sakallah. GRASP–a new search algorithm for satisfiability. In *Proc. IEEE/ACM Int'l Conf. on Computer-Aided Design*, pages 220–227. IEEE Comput. Soc. Press, 1996.

[VB01] M. N. Velev and R. E. Bryant. EVC: A validity checker for the logic of equality with uninterpreted functions and memories, exploiting positive equality and conservative transformations. In *Computer-Aided Verification (LNCS 2102)*, pages 235–240. Springer-Verlag, July 2001.

[VG02a] Allen Van Gelder. Extracting (easily) checkable proofs from a satisfiability solver that employs both preorder and postorder resolution. In *Seventh Int'l Symposium on AI and Mathematics*, Ft. Lauderdale, FL, 2002. (Also available at `ftp://ftp.cse.ucsc.edu/pub/avg/CBJ/sat-pre-post.ps.gz`.).

[VG02b] Allen Van Gelder. Generalizations of watched literals for backtracking search. In *Seventh Int'l Symposium on AI and Mathematics*, Ft. Lauderdale, FL, 2002. (Also available at `ftp://ftp.cse.ucsc.edu/pub/avg/CBJ/watched-lits.ps.gz`.).

[VGO99] A. Van Gelder and F. Okushi. Lemma and cut strategies for propositional model elimination. *Annals of Mathematics and Artificial Intelligence*, 26(1–4):113–132, 1999.

[VGT96] A. Van Gelder and Y. K. Tsuji. Satisfiability testing with more reasoning and less guessing. In D. S. Johnson and M. Trick, editors, *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge.*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society, 1996. (also at `ftp://ftp.cse.ucsc.edu/pub/avg/kclose-tr.ps.Z`).

[Zha97] H. Zhang. SATO: An efficient propositional prover. In *14th International Conference on Automated Deduction*, pages 272–275, 1997.

[ZMMM01] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *ICCAD*, Nov. 2001.