A Tutorial on Satisfiability Modulo Theories*

(Invited Tutorial)

Leonardo de Moura[1], Bruno Dutertre[2], and Natarajan Shankar[2]

[1] Microsoft Research
1 Microsoft Way,
Redmond WA 98052 USA
leonardo@microsoft.com
http://research.microsoft.com/leonardo/
[2] Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA
{bruno,shankar}@csl.sri.com
http://www.csl.sri.com/~{bruno,shankar}/

Abstract. Solvers for satisfiability modulo theories (SMT) check the satisfiability of first-order formulas containing operations from various theories such as the Booleans, bit-vectors, arithmetic, arrays, and recursive datatypes. SMT solvers are extensions of Boolean satisfiability solvers (SAT solvers) that check the satisfiability of formulas built from Boolean variables and operations. SMT solvers have a wide range of applications in hardware and software verification, extended static checking, constraint solving, planning, scheduling, test case generation, and computer security. We briefly survey the theory of SAT and SMT solving, and present some of the key algorithms in the form of pseudocode. This tutorial presentation is primarily directed at those who wish to build satisfiability solvers or to use existing solvers more effectively.

1 Introduction

Satisfiability is the basic and ubiquitous problem of determining if a formula expressing a constraint has a model or a solution. A large number of problems can be described in terms of satisfiability, including graph problems, puzzles such as Sudoku, planning, scheduling, software and hardware verification, extended static checking, optimization, test case generation, among others. Many of these problems can be encoded by Boolean formulas and solved using Boolean satisfiability (SAT) solvers. Other problems require the added expressiveness of equality, uninterpreted function symbols, arithmetic, arrays, datatype operations, and quantifiers. Such problems can be handled by solvers for theory satisfiability or satisfiability modulo theories (SMT). In recent years, satisfiability procedures have undergone dramatic improvements in efficiency and expressiveness. SAT solvers like WalkSAT [SKC96], SATO [Zha97],

* This research was supported NSF Grants CCR-ITR-0326540 and CCR-ITR-0325808. We thank Sam Owre and Ashish Tiwari for their comments and corrections.

W. Damm and H. Hermanns (Eds.): CAV 2007, LNCS 4590, pp. 20–36, 2007.

GRASP [MSS99], Chaff [MMZ+01], zChaff [ZM02,Zha03], Siege [Rya04], and MiniSAT [ES03] have introduced several enhancements to the efficiency of SAT solving. Though SMT technology has been in development since the late 1970s with the work of Shostak [Sho79] and Nelson and Oppen [NO79,Nel81], the incorporation of SAT-based search has yielded very significant efficiencies. Satisfiability is an active and growing area of research with a number of exciting applications and connections to artificial intelligence, operations research, and computational biology. The present tutorial is mostly based on the Yices SMT solver [DdM06b]. It is directed at non-experts and aims to explain some of the basic principles of SAT and SMT solving.

Section 2 covers the basic background on logic and satisfiability. In Section 3, we explain the basic DPLL search procedures for satisfiability. Procedures for solving constraints in individual theories are discussed in Section 4. Theory combinations are discussed in Section 5, and the DPLL-based search procedure for satisfiability modulo theories is presented in Section 6. E-graph matching [Nel81,DNS03] described in Section 7 is an important technique for introducing relevant instantiations of quantified formulas within a search procedure.

2 Preliminaries

We explain the basic syntactic and semantic background needed to follow the rest of the tutorial.

2.1 Propositional Logic

A propositional formula ϕ can be a propositional variable p or a negation $\neg\phi_0$, a conjunction $\phi_0 \wedge \phi_1$, a disjunction $\phi_0 \vee \phi_1$, or an implication $\phi_0 \Rightarrow \phi_1$ of smaller formulas ϕ_0, ϕ_1. A truth assignment M for a formula ϕ maps the propositional variables in ϕ to $\{\top, \bot\}$. A given formula ϕ is *satisfiable* if there is a truth assignment M such that $M \models \phi$ under the usual truth table interpretation of the connectives. If $M \models \phi$ for every truth assignment M, then ϕ is valid. A propositional formula is either valid or its negation is satisfiable.

A literal is either a propositional variable p or its negation $\neg p$. The negation of a literal p is $\neg p$, and the negation of $\neg p$ is just p. A formula is a clause if it is the iterated disjunction of literals of the form $l_1 \vee \ldots \vee l_n$ for literals l_i, where $1 \leq i \leq n$. A formula is in conjunctive normal form (CNF) if it is the iterated conjunction of clauses $\Gamma_1 \wedge \ldots \wedge \Gamma_m$ for clauses Γ_i, where $1 \leq i \leq m$.

2.2 First-Order Logic

In defining a first-order signature, we assume countable sets of variables X, function symbols $\mathcal{F}$, and predicates $\mathcal{P}$. A first-order logic signature Σ is a partial map from $\mathcal{F} \cup \mathcal{P}$ to the natural numbers corresponding to the *arity* of the symbol. A *Σ-term* τ has the form

$$\tau := x \mid f(\tau_1, \ldots, \tau_n),$$

where $f \in \mathcal{F}$ and $\Sigma(f) = n$. For example, if $\Sigma(f) = 2$ and $\Sigma(g) = 1$, then $f(x, g(x))$ is a Σ-term. A *Σ-formula* has the form

$$\psi := p(\tau_1, \ldots, \tau_n) \mid \tau_0 = \tau_1 \mid \neg\psi_0 \mid \psi_0 \vee \psi_1 \mid \psi_0 \wedge \psi_1 \mid (\exists x : \psi_0) \mid (\forall x : \psi_0),$$

where $p \in \mathcal{P}$ and $\Sigma(p) = n$, and each τ_i, $1 \leq i \leq n$ is a Σ-term. For example, if $\Sigma(<) = 2$ for a predicate symbol $<$, then $(\forall x : (\exists y : x < y))$ is a Σ-formula. The set of free variables in a formula ψ is represented as $vars(\psi)$. A *sentence* is a formula with no free variables.

A Σ-structure M consists of a nonempty domain $|M|$ where for each $f \in F$ such that $\Sigma(f) = n$, $M(f)$ is an n-ary map on $|M|$, for each $p \in \mathcal{P}$ such that $\Sigma(p) = n$, $M(p)$ is a subset of $|M|^n$, and for each $x \in X$, $M(x) \in |M|$. The interpretation of a term a in M is given by $M[\![x]\!] = M(x)$ and $M[\![f(a_1, \ldots, a_n)]\!] = M(f)(M[\![a_1]\!], \ldots, M[\![a_n]\!])$. For a Σ-formula ψ and a Σ-structure M, satisfaction $M \models \psi$ can be defined as

$$\begin{aligned}
M \models a = b &\iff M[\![a]\!] = M[\![b]\!] \\
M \models p(a_1, \ldots, a_n) &\iff (M[\![a_1]\!], \ldots, M[\![a_n]\!]) \in M(p) \\
M \models \neg\psi &\iff M \not\models \psi \\
M \models \psi_0 \vee \psi_1 &\iff M \models \psi_0 \text{ or } M \models \psi_1 \\
M \models \psi_0 \wedge \psi_1 &\iff M \models \psi_0 \text{ and } M \models \psi_1 \\
M \models (\forall x : \psi) &\iff M\{x \mapsto \mathbf{a}\} \models \psi, \text{ for all } \mathbf{a} \in |M| \\
M \models (\exists x : \psi) &\iff M\{x \mapsto \mathbf{a}\} \models \psi, \text{ for some } \mathbf{a} \in |M|
\end{aligned}$$

A first-order Σ-formula ψ is satisfiable if there is a Σ-structure M such that $M \models \psi$, and it is valid if in all Σ-structures M, $M \models \psi$. A Σ-sentence is either satisfiable or its negation is valid. We focus on the satisfiability problem for quantifier-free first-order formulas.

3 SAT Solving

The principles of modern SAT solving have their origin in the 1960 procedure of Davis and Putnam [DP60], as simplified in 1962 by Davis, Logemann, and Loveland [DLL62]. The first step in the Davis–Putnam–Logemann–Loveland (DPLL) procedure is to convert the formula to conjunctive normal form (CNF) by introducing new variables to label the subformulas. A formula can be converted to clausal form by introducing fresh variables for each compound subformula and adding suitable clauses, e.g., in converting $\neg p \vee (\neg q \wedge r)$, we label $\neg q \wedge r$ as b and $\neg p \vee b$ as a to obtain the clauses $a, a \vee p, a \vee \neg b, \neg a \vee \neg p \vee b, b \vee q \vee \neg r, \neg b \vee \neg q, \neg b \vee r$.

The input to the satisfiability procedure is given as a set of clauses K representing the CNF formula $\bigwedge K$. The DPLL procedure builds a *partial truth assignment* for the variables in K by successively guessing an assignment for an unassigned literal, propagating the consequences of the partial assignment with respect to the clauses, and backtracking on the partial assignment when a conflict is detected in the form of a falsified clause. The procedure terminates either with a truth assignment satisfying each of

$$
\begin{array}{rll}
\mathit{dpll}(K) := & \mathit{dpllr}(0, \emptyset, K, \emptyset) & (\mathit{init}) \\
\mathit{dpllr}(0, M, K, C) := & \bot, \text{ if} & (\mathit{contrad}) \\
& \mathit{propagate}(M, K, C) = \bot[\Gamma] & \\
\mathit{dpllr}(h+1, M, K, C) := & \mathit{dpllr}(h', M_{h'}, K, C'), \text{ where} & (\mathit{backjump}) \\
& \mathit{propagate}(M, K, C) = \bot[\Gamma], & \\
& \mathit{analyze}(h+1, M, \Gamma) = \Gamma', & \\
& C' = C \cup \{\Gamma'\}, & \\
& h' = L2(\Gamma') & \\
\mathit{dpllr}(h, M, K, C) := & \mathit{dpllr}(h+1, M'', K, C), \text{ where} & (\mathit{split}) \\
& M' = \mathit{propagate}(M, K, C) \neq \bot, & \\
& l = \mathit{select}(M', K) \neq \bot, & \\
& M'' = M'; l & \\
\mathit{dpllr}(h, M, K, C) := & M', \text{ where} & (\mathit{sat}) \\
& M' = \mathit{propagate}(M, K, C) \neq \bot, & \\
& \mathit{select}(M', K) = \bot &
\end{array}
$$

Fig. 1. The DPLL Boolean Satisfiability Procedure

the clauses, or with a demonstration that no such assignment can be constructed. The state of the search procedure is a 4-tuple $\langle h, M, K, C\rangle$ consisting of the *decision level* h, the partial assignment M, the input clause set K, and a set C of *conflict clauses* derived from K that are constructed during the search.

At a decision level h, the partial assignment consists of a sequence $M_0; \ldots; M_h$. Each M_i at decision level i is of the form $d; \langle l_1[\Gamma_1], \ldots, l_k[\Gamma_k]\rangle$ for some k, where d is the *decision literal* at level i, and each l_i is an *implied literal* and the clause Γ_i occurs in $K \cup C$. The assignment M_0 contains no decision literal. A decision literal or implied literal in M is said to be an assigned literal in M. No assigned literal in M occurs twice in M, nor does it occur negated in M. The assignment corresponding to M maps a variable p to $\top$ (respectively, $\bot$) if p (respectively, $\neg p$) is an assigned literal in M. If neither p nor $\neg p$ occurs in M, then the assignment is undefined. Given an assigned literal l occurring in M at level i, the assignment preceding l, written as $M_{<l}$, consists of $M_0; \ldots; M_{i-1}; M_i^{<l}$, where $M_i^{<l}$ consists of the part of the assignment of M_i preceding the occurrence of l. For each entry $l[\Gamma]$ in M, the clause Γ occurs in $K \cup C$ and is of the form $l \vee \Gamma'$, where $M_{<l} \models \neg\Gamma'$. The notation $M_{\overline{h}}$ represents the sequence $M_0; \ldots; M_h$.

The DPLL search algorithm shown in Figure 1 works by constructing the partial assignment M through the use of *propagation*, *analysis/backjumping*, and decision literal *selection*, until it has constructed an assignment satisfying the input clauses K or it can be shown that there is no such assignment. For decision level $h > 0$, the propagation operation $\mathit{propagate}(h, M, K, C)$ shown in Figure 2 works by adding $l[\Gamma]$ to M_h, where $\Gamma \in K \cup C$ is of the form $l \vee \Gamma'$, where $M \models \neg\Gamma'$. When $h = 0$, each unit clause l in $K \cup C$ is placed in M_0 as $l[l]$. Propagation can also detect a conflict where there is a clause of the form Γ such that $M \models \neg\Gamma$. If a conflict is detected at decision level 0, then the dpll algorithm reports unsatisfiability. If no conflict is detected, then a literal that is unassigned in M is selected using the $\mathit{select}(M, K)$ operation and added to the partial assignment. The procedure is then invoked at level $h + 1$.

$$
\begin{array}{lll}
propagate(M,K,C) := propagate(\langle M, l[\Gamma]\rangle, K, C), \text{ where} & & (unit)\\
\quad \Gamma \in K \cup C, & & \\
\quad \Gamma \equiv l \vee l_1 \vee \ldots \vee l_n, & & \\
\quad M \not\models l, & & \\
\quad M \models \neg l_i \wedge \ldots \wedge \neg l_n & & \\
propagate(M,K,C) := \bot[\Gamma], \text{ where} & & (conflict)\\
\quad \text{if } \Gamma \in K \cup C : M \models \neg\Gamma & & \\
propagate(M,K,C) := M, \text{ where} & & (terminate)\\
\quad \text{for each } \Gamma \in K \cup C, & & \\
\quad M \models \Gamma \text{ or} & & \\
\quad \Gamma \equiv l \vee l' \vee \Gamma', \text{ and } l, l' \notin dom(M) & &
\end{array}
$$

Fig. 2. DPLL Propagation

Otherwise, if clause Γ in $K \cup C$ is the source of the conflict, it can be analyzed by the $analyze(h, M, \Gamma)$ operation to construct a conflict clause that is added to C. Here, Γ is of the form $l_1 \vee \ldots \vee l_n$ where M contains $\neg l_i[\neg l_i \vee \Gamma_i]$, for $1 \leq i \leq n$. The analysis phase successively replaces Γ with the result of resolving Γ with each clause $\neg l_i \vee \Gamma_i$ for l_i occurring at level h until Γ contains a unique literal l at level h. Note that $M \models \neg\Gamma$ for each such clause Γ generated through analysis. Furthermore, the clause Γ contains at least one literal l such that $\neg l$ is assigned at level h since the conflict is detected at level h. The analysis process is iterated until there is a unique such literal l such that $M_h \models \neg l$. The clause $\Gamma' = analyze(h, M, \Gamma)$ constructed by the analysis phase is added as a conflict clause to C to obtain the new conflict clause set C'. Let $h' = L2(\Gamma')$ be the highest level below h such that there is a literal l' in Γ' with $M_{h'} \models \neg l'$. The unique literal l at level h in Γ' is implied by the partial assignment $M_{\overline{h'}}$ and Γ'.

The search is resumed with the state $\langle h', M_{\overline{h'}}, K, C'\rangle$. Though the partial assignment has shrunk, it now contains more implied literals at level h'. On the other hand, if no conflict is detected at level h, then an unassigned literal d is *selected* as the decision literal at level $h+1$, and the search is resumed with the state $\langle h+1, \langle M; d\rangle, K, C\rangle$. If no unassigned literals remain, then the algorithm terminates with a satisfying assignment M for K. Termination [NOT06,Sha05] follows since each step of propagation, backjumping (with propagation), or selection increases the quantity $\sum_{i=0}^{h} |M_i| * N^{(N-h)}$ towards the bound $N^{(N+1)}$ for $N = |vars(K)|$.

We have assumed that the propagation phase is complete, but the procedure works even when the propagation step is incomplete so that M_h need not contain all the literals that are implied by $M_0; \ldots; M_h$. Thus it is possible that M_j contains literals that are actually implied at some level i, with $i < j$. In this case, a conflict can still be traced to some level $\hat{h}$ below the current level h, and the $analyze$ operation can be modified to construct a conflict clause Γ that contains a unique literal at level $\hat{h}$.

The algorithm can either terminate with an assignment M satisfying the input clause set K, or with an unsatisfiability when a conflict is reported at the decision level 0. The SAT procedure can also generate a proof of unsatisfiability since a conflict at level 0 implies that some clause Γ in K when resolved with other clauses from $K \cup C$ yields a contradiction. The clauses in C are themselves derived by resolution.

step	h	M	K	C	Γ
select s	1	$;s$	K	$\emptyset$	_
select r	2	$;s;r$	K	$\emptyset$	_
propagate	2	$;s;r,\neg q[\neg q \vee \neg r]$	K	$\emptyset$	_
propagate	2	$;s;r,\neg q,p[p \vee q]$	K	$\emptyset$	_
conflict	2	$;s;r,\neg q,p$	K	$\emptyset$	$\neg p \vee q$
analyse	0	$\emptyset$	K	q	_
propagate	0	$q[q]$	K	q	_
propagate	0	$q,p[p \vee \neg q]$	K	q	_
propagate	0	$q,p,r[\neg p \vee r]$	K	q	_
conflict	0	q,p,r	K	q	$\neg q \vee \neg r$

Fig. 3. Example of the DPLL Satisfiability Procedure

Example 1. An example computation of the DPLL algorithm for demonstrating the unsatisfiability of the input K given by $\{p \vee q, \neg p \vee q, p \vee \neg q, s \vee \neg p \vee q, \neg s \vee p \vee \neg q, \neg p \vee r, \neg q \vee \neg r\}$. is shown in Figure 1. In this example, there are no unit input clauses. The partial assignment M_0 is therefore empty. The literal s is selected as the decision literal at level 1. Propagation does not yield any new implied literals at level 1. Then, literal r is selected as the decision literal at level 2. Now propagation adds the literals $\neg q$ and p, but then detects the conflict with clause $\neg p \vee q$. Analyzing this conflict, we obtain the conflict clause q which is added to C while backjumping to level 0. Now there is a unit clause q, and propagation adds the literals p and r to M_0 before detecting the conflict on the clause $\neg q \vee \neg r$. Since this conflict is at level 0, the input clause is judged to be unsatisfiable.

The proof of unsatisfiability for the example in Figure 1 can be constructed by resolution. The conflict clause q is proved by resolving $\neg p \vee q$ with $p \vee q$. The proof of unsatisfiability is constructed by resolving $\neg q \vee \neg r$ with $\neg p \vee r$ to obtain $\neg p \vee \neg q$ which is in turn resolved with $p \vee \neg q$ to obtain $\neg q$ which is resolved with the conflict clause q to derive $\bot$.

Given two clause sets K_1 and K_2 such that $K_1 \cup K_2$ is unsatisfiable, a *Craig interpolant* [Cra57] is a formula ϕ whose propositional variables appear in both K_1 and K_2 such that $K_1 \implies \phi$ and ϕ, K_2 is unsatisfiable. Interpolants are useful for a number of applications and can be extracted from proofs of unsatisfiability [McM05]. The DPLL procedure can also be used for computing the disjunctive normal form (DNF) or Binary Decision Diagram (BDD) representation corresponding to all satisfying assignments for a formula. The DPLL procedure can also be used to construct a minimal unsatisfiable core of clauses from the input and a maximal subset of the input clauses that is satisfiable. Pseudo-Boolean constraints are of the form $\sum_{i=1}^{n} c_i * p_i \geq N$, where for $1 \leq i \leq n$, c_i is an integer constant, N is an integer constant, and $c_i * p_i$ equals c_i if p_i, and 0, otherwise. The conjunction of a clause set K together with pseudo-Boolean constraints can also be solved.

SAT solving can be used to solve constraints over finite domains involving planning and scheduling, and in the verification of finite-state hardware and software systems. Key ideas in the development of efficient SAT solvers originate from SATO [Zha97],

GRASP [MSS99], and Chaff [MMZ+01]. Efficient implementations of SAT algorithms include ZChaff [ZM02,Zha03], Berkmin [GN02], Siege [Rya04], and MiniSAT [ES03].

4 Theory Constraint Solving

We now examine satisfiability in first-order theories. These theories can be presented axiomatically or as a class of first-order structures. We define a theory over a signature Σ as a class of first-order structures closed under isomorphism and variable reassignment. The current section will examine the clausal validity problem (CVP) of determining if a clause $l_1 \vee \ldots \vee l_n$ is valid, or equivalently, if the conjunction $\neg l_1 \wedge \ldots \wedge \neg l_n$ is satisfiable. For SMT applications, it is important that these procedures support the incremental assertion of literals, efficient backtracking, and the production of explanations in the form of the subset of input literals needed for unsatisfiability.

4.1 Equivalence

CVP for an equivalence relation given by axioms for reflexivity, symmetry, and transitivity can be solved using the union-find algorithm. The input literals are equalities and disequalities between variables. The algorithm maintains two data structures: a mapping F on the variables in the input and a set of input disequalities D. The *find structure* F must be acyclic so that for any $n > 0$ and variable x, either $F(x) = x$ or $F^n(x) \neq x$. The operation $F^*(x)$ can be defined to return the *canonical* representative of the equivalence class containing x. The operation $union(F)(x, y)$ is used to construct the find structure in which the equivalence classes of x and y are merged. It assumes a total ordering $\prec$ on the variables.

$$union(F)(x,y) = \begin{cases} F[x' := y'], y' \prec x' \\ F[y' := x'], otherwise \end{cases}$$
$$\text{where } x' \equiv F^*(x) \not\equiv F^*(y) \equiv y'$$

The $addeqlit$ procedure shown in Figure 4 takes as input a literal l (an equality or disequality), the find structure F, and the disequality set D. Initially $F(x) = x$ for each variable x, and D is empty. The operation $addeqlit(l, F, D)$ updates the *state* $\langle F, D\rangle$ with the constraint given by the literal l.

There are many variations on this basic theme that involve path compression and tree-weight directed union which together yield the near-linear $O((m + n) * \alpha(n))$ complexity for m union/find operations over n variables [GF64,Tar75]. The algorithm can also be augmented to maintain proofs in the form of transitivity chains and to support efficient retraction [dMRS04,NO05]. The algorithm can be applied to equivalence relations other than equality.

4.2 Congruence Closure

The free theory $\Phi(\Sigma)$ over a signature Σ is the first-order theory with an empty set of non-logical axioms. Equality is treated as a logical symbol with the axioms of reflexivity, symmetry, transitivity, and congruence. Note that the equivalence theory above is

$$
\begin{array}{lll}
addeqlit(x = y, F, D) := \langle F, D\rangle, \text{ if} & & (skip)\\
\quad F^*(x) \equiv F^*(y) & & \\
addeqlit(x = y, F, D) := \begin{cases} \bot, \text{ if } F'^*(u) \equiv F'^*(v) \text{ for some } u \neq v \in D \\ \langle F', D\rangle, \text{otherwise} \end{cases} & & (union)\\
\quad \text{where} & & \\
\quad x' = F^*(x) \not\equiv F^*(y) = y', & & \\
\quad F' = union(F)(x, y) & & \\
addeqlit(x \neq y, F, D) := \bot, \text{ if } F^*(x) \equiv F^*(y) & & (contrad)\\
addeqlit(x \neq y, F, D) := \langle F, D\rangle, \text{ if } F^*(x) \equiv F^*(x'), F^*(y) \equiv F^*(y'), & & (skipdiseq)\\
\quad \text{for } x' \neq y' \in D & & \\
addeqlit(x \neq y, F, D) := \langle F, \{x \neq y\} \cup D\rangle, \text{ otherwise.} & & (adddiseq)
\end{array}
$$

Fig. 4. Adding an Equality to a Union-Find Structure

just the free theory $\Phi(\emptyset)$ over an empty signature. CVP for $\Phi(\Sigma)$ requires the extension of the union–find procedure to the computation of the congruence closure [Koz77,Sho78]. Bachmair, Tiwari, and Vigneron [BTV03] give an elegant presentation of congruence closure in the form of inference rules. Our presentation is closer to a typical implementation.

A congruence relation extends equivalence with the rule that for each n-ary function f, $f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n)$ if $s_i = t_i$ for each $1 \leq i \leq n$. The operation $congruent(F, s, t)$ checks if $s \equiv f(s_1, \ldots, s_n)$, $t \equiv f(t_1, \ldots, t_n)$ such that $F^*(s_i) \equiv F^*(t_i)$ for $1 \leq i \leq n$. The term universe T which includes every term in the CVP is assumed to be given. Any subterm of a term in T is also a member of T. For any term t in T, $\pi(t)$ returns the set of terms t' in T of the form $f(t_1, \ldots, t_n)$ such that for some i, $1 \leq i \leq n$, $t \equiv t_i$. The congruence closure operation for closing a find structure under congruence is shown in Figure 5.

$$
\begin{array}{lll}
close(F, D, Q, \pi) := close(F', D, Q', \pi), & & (congruence)\\
\quad \text{when } s, t : s = t \in Q, F^*(s) \not\equiv F^*(t), & & \\
\quad congruent(F, s, t) & & \\
\quad \langle F', D, \pi'\rangle = addeqlit(s = t, F, D, Q, \pi), & & \\
\quad Q' = Q \cup \{s' = t' | \begin{array}{l} s' \in \pi(s), t' \in \pi(t), \\ congruent(F', s', t')\} \end{array} & & \\
close(F, D, Q, \pi) := \langle F, D, \pi\rangle, \text{ otherwise.} & & (terminate)
\end{array}
$$

Fig. 5. Congruence Closure

The $addeqlit$ operation can be modified to make use of $close$, and the only relevant case of this definition is shown below.

$$
\begin{array}{l}
addeqlit(s = t, F, D, \pi) := \begin{cases} \bot, \text{ if } F'(u) \equiv F'(v) \text{ for some } u \neq v \in D \\ close(F', D, \emptyset, \pi), \text{ otherwise} \end{cases}\\
\quad \text{where}\\
\quad s' = F^*(s), t' = F^*(t), s' \not\equiv t',\\
\quad s' \prec t', F' = F[t' := s'], \pi' = \pi[s' := \pi(s') \cup \pi(t')]
\end{array}
$$

The *Ackermann reduction* is a simple alternative to congruence closure. It works by reducing congruence to equivalence by successively replacing each term $f(\overline{x})$ in the given formula ψ with a fresh variable $x_{f(\overline{x})}$ to obtain ψ'. The satisfiability of ψ is equivalent to that of $\psi' \wedge \bigwedge\{x_1 \neq y_1 \vee \ldots \vee x_n \neq y_n \vee x_{f(\overline{x})} = x_{f(\overline{y})} \mid x_{f(\overline{x})}, y_{f(\overline{x})} \in vars(\psi')\}$, and the latter formula is in $\Phi(\emptyset)$.

4.3 Difference Arithmetic

Difference arithmetic (DA) deals with arithmetic constraints of the form $x - y \leq c$, where c is an integer constant. Equality constraints $x = y$ can be expressed as $x - y \leq 0 \wedge y - x \leq 0$. Strict inequalities can also be captured so that $x - y < c$ is just $x - y \leq c - 1$, and the negation of $x - y \leq c$ is just $y - x \leq -c - 1$. By introducing a special variable x_0 representing 0, we can also express unary constraints of the form $x \leq c$ as $x - x_0 \leq c$ and $x \geq c$ as $x_0 - x \leq -c$.

A conjunction of such constraints is satisfiable if there is an assignment ρ of integers to the variables such that for each inequality $x - y \leq c$, the integer difference $\rho(x) - \rho(y)$ evaluates to a value that is at most c.

Difference constraints can be modeled by means of a weighted directed graph with the variables as vertices with an edge of weight c from y to c corresponding to each constraint $x - y \leq c$. The Bellman–Ford algorithm can be employed in an incremental form to find an integer assignment, when one exists, for the variables satisfying the constraints. If there is a negative-weight cycle of edges such that $x - x_1 \leq c_1, x_1 - x_2 \leq c_2, \ldots x_n - x \leq c_n$, where $\sum_{i=1}^{n} c_n < 0$, then the constraints are unsatisfiable since there is no assignment to x or the other variables in the cycle that would satisfy the chaining of these inequalities.

The procedure maintains a variable assignment ρ so that it satisfies the inequality constraints processed, and an edge map E such that for each vertex y, $E(y)$ is a set of pairs $\langle x, c\rangle$ such that $x - y \leq c$ is a constraint that has been processed. Initially, the assignment ρ can be arbitrary, and the edge map E is empty. The $addineq(x, y, c, \rho, E)$ operation adds a constraint $x - y \leq c$ to $\langle \rho, E\rangle$.

$$
\begin{aligned}
addineq(x, y, c, \rho, E) := {} & \langle \rho, E[y := E(y) \cup \{\langle x, c\rangle\}]\rangle, \text{ if} \\
& \rho(x) - \rho(y) \leq c \\
addineq(x, y, c, \rho, E) := {} & \begin{cases} \bot, \text{ if } \rho'' = \bot \\ \langle \rho'', E'\rangle, \text{ otherwise} \end{cases} \\
& \text{where } \rho(x) - \rho(y) > c \\
& \rho' = \rho[x := \rho(y) + c] \\
& E' = E[y := E(y) \cup \{\langle x, c\rangle\}], \\
& \rho'' = relaxv(y, \rho', E', \{x\})
\end{aligned}
$$

The operation $relaxv(y, \rho, E, Q)$ is defined to *relax* each edge $\langle x, z\rangle$ by ensuring that $\rho(z) - \rho(x) \leq c$ for $\langle z, c\rangle \in E(x)$. If the vertex y itself appears in the queue of vertices to be processed, then a negative weight cycle is signaled.

$$
\begin{aligned}
\mathit{relaxv}(y, \rho, E, \emptyset) &:= \rho \\
\mathit{relaxv}(y, \rho, E, Q) &:= \bot, \text{ if } y \in Q \\
\mathit{relaxv}(y, \rho, E, Q) &:= \mathit{relaxv}(y, \rho', E, Q'), \text{ where} \\
&\qquad \langle \rho', Q' \rangle = \mathit{relax}(x, \rho, Q), \text{ for } x \in Q \\
\mathit{relax}(x, \rho, Q) &:= \langle \rho', Q' \rangle, \text{ where} \\
&\qquad Q' = (Q - \{x\}) \cup \{z \mid \langle z, c \rangle \in E(x), \rho(z) - \rho(x) > c\} \\
&\qquad \rho' = \rho \circ [z \mapsto \rho(x) + c \mid \langle z, c \rangle \in E(x), \rho(z) - \rho(x) > c]
\end{aligned}
$$

The above incremental procedure is based on the incremental Bellman–Ford algorithm of Wang, Ivančić, Ganai, and Gupta [WIGG05]. Cherkassky and Goldberg [CG96] give a survey of negative-weight cycle detection algorithms.

4.4 Linear Arithmetic

Linear arithmetic (LA) constraints have the form $c_0 + \sum_{i=1}^{n} c_i * x_i \leq 0$, where each c_i, for $0 \leq i \leq n$ is a rational constant, and the variables x_i range over the reals. The LA solver described below is based on the method of de Moura and Dutertre [DdM06a]. This method is often faster than the Bellman–Ford procedure on difference arithmetic constraints, and supports an efficient but incomplete check for unsatisfiability that is useful in an SMT solver.

The input to the procedure is

- A set of n real-valued variables $x_1, \ldots, x_n$
- A set of m linear equalities (where $m \leq n$)

$$
\begin{aligned}
a_{11}x_1 + \ldots + a_{1n}x_n &= 0 \\
&\vdots \\
a_{m1}x_1 + \ldots + a_{mn}x_n &= 0
\end{aligned}
$$

  written in matrix form, $Ax = 0$.
- Bounds on all variables: $l_i \leq x_i \leq u_i$ where l_i is either $-\infty$ or a rational number, and u_i is either $+\infty$ or a rational number.

The goal is to determine whether there is x such that $Ax = 0$ and $l_i \leq x_i \leq u_i$ for $i = 1, \ldots, n$ (i.e., whether the constraints are satisfiable).
The solver maintains a *tableau* and an *assignment*.

- The tableau is defined by dividing the variables into a set B of m basic variables and a set N of $n - m$ non-basic variables, then rewriting the constraints $Ax = 0$ as follows:

$$
\begin{aligned}
x_{i_1} &= \sum_{x_j \in N} b_{1j} x_j \\
&\vdots \\
x_{i_m} &= \sum_{x_j \in N} b_{mj} x_j
\end{aligned}
$$

  where $x_{i_1}, \ldots, x_{i_m}$ are the basic variables.

– The assignment β assigns a rational value $\beta(x_i)$ to every variable x_i, such that
  - For all non-basic variable x_j, we have $l_j \leq \beta(x_j) \leq u_j$.
  - For all basic variable x_{i_k}, we have

$$\beta(x_{i_k}) = \sum_{x_j \in N} b_{kj}\beta(x_j).$$

If β also satisfies the bounds on basic variables, namely,

$$l_{i_k} \leq \beta(x_{i_k}) \leq u_{i_k}$$

for $k = 1, \ldots, n$ then the constraints are satisfiable and $\beta(x_1)...\beta(x_n)$ is a feasible solution. Otherwise, if there is a basic variable x_{i_k} with $\beta(x_{i_k}) < l_{i_k}$, then a pivoting step is used to swap it with a non-basic variable x_j such that $b_{kj} > 0$ and $x_j < u_j$ or $b_{kj} < 0$ and $x_j > l_j$, and symmetrically when $\beta(x_{i_k}) > u_{i_k}$.

$A_0 = \begin{cases} s_1 = -x + y \\ s_2 = x + y \end{cases}$		$\beta_0 = (x \mapsto 0, y \mapsto 0, s_1 \mapsto 0, s_2 \mapsto 0)$
$A_1 = A_0$	$x \leq -4$	$\beta_1 = (x \mapsto -4, y \mapsto 0, s_1 \mapsto 4, s_2 \mapsto -4)$
$A_2 = A_1$	$-8 \leq x \leq -4$	$\beta_2 = \beta_1$
$A_3 = \begin{cases} y = x + s_1 \\ s_2 = 2x + s_1 \end{cases}$	$-8 \leq x \leq -4$ $s_1 \leq 1$	$\beta_3 = (x \mapsto -4, y \mapsto -3, s_1 \mapsto 1, s_2 \mapsto -7)$

Fig. 6. Example

Figure 6 illustrates the algorithm on a small example. Each row represents a state. The columns contain the tableaux, bounds, and assignments. The first row contains the initial state. Suppose $x \leq -4$ is asserted. Then the value of x must be adjusted, since $\beta_0(x) > -4$. Since s_1 and s_2 depend on x, their values are also modified. No pivoting is required since the basic variables do not have bounds, so $A_1 = A_0$. Next, $x \geq -8$ is asserted. Since $\beta_1(x)$ satisfies this bound, nothing changes: $A_2 = A_1$ and $\beta_2 = \beta_1$. Next, $s_1 \leq 1$ is asserted. The current value of s_1 does not satisfy this bound, so s_1 is pivoted with y to decrease s_1. The resulting state S_3 is shown in the last row; all constraints are satisfied.

If $s_2 \geq -3$ is asserted in S_3 then an inconsistency is detected: Tableau A_2 does not allow s_2 to increase since both x and s_1 are at their upper bound. Therefore, $s_2 \geq -3$ is inconsistent with state S_3.

5 Combining Theories

We have shown solutions to the CVP problem for individual theories such as linear arithmetic and the theory of equality over uninterpreted terms. Many natural constraint solving problems contain symbols from multiple theories. Given two theories $\mathcal{T}_1$ and $\mathcal{T}_2$

over signatures Σ_1 and Σ_2, the union theory $\mathcal{T}_1 + \mathcal{T}_2$ is the class of Σ-structures, with $\Sigma = \Sigma_1 \cup \Sigma_2$ whose projection to Σ_i is a $\mathcal{T}_i$-model, for $i = 1, 2$. The easiest case to consider is when Σ_1 and Σ_2 are disjoint. The Nelson–Oppen procedure [NO79] gives a method for composing CVP-solvers for $\mathcal{T}_1$ and $\mathcal{T}_2$ into one for $\mathcal{T}_1 + \mathcal{T}_2$. For example, $\mathcal{T}_1$ the free theory $\Phi(\Sigma_1)$ over a signature Σ and $\mathcal{T}_2$ is the difference arithmetic theory DA.

A quantifier-free Σ-formula ψ can be *purified* so that each literal in the formula is a Σ_i-literal for $i = 1, 2$. Let $\psi[t := s]$ be the result of replacing each occurrence of t in ψ by s. A *pure* Σ_i-term, for $i = 1, 2$, is a Σ_i-term that is not a variable.

$$\begin{aligned} \mathit{purify}(\psi, R) := \; & \mathit{purify}(\psi[t := x], R \cup \{x = t\}), \\ & \text{for fresh } x, \\ & \text{pure } \Sigma_i\text{-term } t \text{ in } \psi, i = 1, 2 \\ \mathit{purify}(\psi, R) := \; & (\bigwedge R) \wedge \psi, \text{ otherwise.} \end{aligned}$$

The main point of purification is that if $\mathit{purify}(\psi, \emptyset) = \psi'$, then ψ and ψ' are equi-satisfiable and each literal in ψ' is a Σ_1-literal or a Σ_2-literal.

A *partition* Π on a set of variables γ is a disjoint collection subsets $\gamma_1, \ldots, \gamma_n$ such that $\bigcup_{i=1}^{n} \gamma_i = \gamma$. Given a partition Π of the form $\gamma_1, \ldots, \gamma_n$, an arrangement A_Π is a union of the set of equalities $\{x = y \mid \text{ for some } i : x, y \in \gamma_i\}$ and the set of disequalities $\{x \neq y \mid \text{ for some } i, j : i \neq j, x \in \gamma_i, y \in \gamma_j\}$.

A Boolean implicant P for a quantifier-free Σ-formula ψ containing the set of literals L is a subset of literals in L such that $\bigwedge P \implies \psi$ in propositional logic. The formula ψ is $\mathcal{T}$-satisfiable if there is an Boolean implicant P of ψ that is $\mathcal{T}$-satisfiable. If $\mathcal{T} = \mathcal{T}_1 \cup \mathcal{T}_2$ for Σ_1-theory $\mathcal{T}_1$ and Σ_2-theory $\mathcal{T}_2$, with $\Sigma_1 \cap \Sigma_2 = \emptyset$, and $P = P_1 \cup P_2$, where each P_i consists of Σ_i-literals and $\gamma = \mathit{vars}(P_1) \cap \mathit{vars}(P_2)$, then P is $\mathcal{T}$-satisfiable if there is an arrangement A_Π of γ such that each $P_i \cup A_\Pi$ is $\mathcal{T}_i$-satisfiable. For this joint satisfiability result to hold, the theories $\mathcal{T}_1$ and $\mathcal{T}_2$ must be stably infinite [Opp80], i.e., if a formula is $\mathcal{T}_i$-satisfiable, it must be satisfiable in a countable model.

We now show how the $\mathcal{T}$-satisfiability of a quantifier-free Σ-formula can be solved by an extension of the *dpll* procedure.

6 Satisfiability Modulo Theories

The extension of the *dpll* satisfiability procedure to $\mathcal{T}$-satisfiability of quantifier-free Σ-formulas employs an oracle for $\mathcal{T}$-satisfiability of implicants. This procedure adds a context S for the incrementally updated state of the theory solver. We assume that there is a procedure $\mathit{assert}(l[\Gamma], S)$ that adds a literal implied by S and the clause $\Gamma \in K \cup C$ to the state S. When the added literal l is a decision literal, we indicate the absence of an implying clause by $l[]$. This procedure need not be complete with respect to detecting unsatisfiability, so we have a complete procedure $\mathit{check}(S)$ which checks if the state S is satisfiable. We also have a third procedure $\mathit{ask}(l, S)$ which is an incomplete test for determining if the result of adding l to state S is unsatisfiable. We assume that all three procedures when they return $\bot$ also return an explanation clause Γ' of the form $l_1 \vee \ldots \vee l_n$ such that $\neg l_i$ is an input literal. We assume that the state S is check-pointed

$$
\begin{array}{rll}
tdpll(K) := & tdpllr(0, \emptyset, \emptyset, K, \emptyset) & (tinit) \\
tdpllr(0, M, S, K, C) := & \bot, \text{ where} & (tcontrad) \\
& scanprop(M, S, K, C) = \bot[\Gamma] & \\
tdpllr(h+1, M, S, K, C) := & tdpllr(h', M, S_{\overline{h'}}, K, C'), \text{ where} & (tbackjump) \\
& scanprop(M, S, K, C) = \bot[\Gamma], & \\
& tanalyze(h+1, M, \Gamma) = \Gamma', & \\
& C' = C \cup \{\Gamma'\}, & \\
& h' = L2(\Gamma') & \\
tdpllr(h, M, S, K, C) := & tdpllr(h+1, M', S'', K, C), \text{ where} & (tsplit) \\
& \langle M', S' \rangle = scanprop(M, S, K, C) \neq \bot, & \\
& l = tselect(M, S, K) \neq \bot, & \\
& S'' = assert(l[], S') & \\
tdpllr(h, M, S, K, C) := & \left\{ \begin{array}{l} S', \text{ if } check(S') \neq \bot \\ tdpllr(h', M, S_{h'}, K, C'), \text{ where} \\ \quad check(S') = \bot[\Gamma], \\ \quad h' = L2(\Gamma), \;\; C' = C \cup \{\Gamma\} \end{array} \right. & (tcheck) \\
& \text{with } S' = scanprop(M, S, K, C) \neq \bot, & \\
& tselect(M, S', K) = \bot &
\end{array}
$$

Fig. 7. DPLL Search for Satisfiability Modulo Theories

at each level so that S_i represents the state at level i including all the input assertions up to that point.

With these, we can modify the $dpll$ procedure from Section 3 as shown in Figure 7. The main difference from $dpll$ is that the selected literal is asserted to the context S and the complete $check$ procedure is used to check for $\mathcal{T}$-satisfiability of the context S when there are no splitting literals left. The procedure of literal selection $tselect$ can be identical to $select$. The theory propagation procedure $scanprop$ is defined below. It first identifies some of the literals l such that l or $\neg l$ appears in K that are entailed by the context S.

$$
\begin{array}{rl}
scanprop(M, S, K, C) := & tpropagate(M', S, K, C'), \text{ where} \\
& \langle M', C' \rangle = scanlits(M, S, K, C) \\
scanlits(M, S, K, C) := & \langle M', C' \rangle, \text{ where} \\
& M' = M \circ \langle l \in lits(K) - lits(M) \mid ask(\neg l, S) = \bot[\Gamma] \rangle, \\
& C' = C \cup \{\Gamma \mid \exists l \in lits(K) - lits(M) : ask(\neg l, S) = \bot[\Gamma]\}
\end{array}
$$

The $tpropagate$ procedure is adapted from the $propagate$ procedure from Section 3 and shown in Figure 8. The literals that are added to M are also asserted to the context S.

Methods combining DPLL SAT solving with theory constraint solving were introduced in CVC [BDS02], ICS [dMRS02], and Verifun [FJOS03], and Nieuwenhuis, Oliveras, and Tinelli [NOT06] give a rigorous and abstract presentation of this combination.

$$tpropagate(M,S,K,C) := \begin{cases} \bot[\Gamma], \text{ if } S' = \bot[\Gamma] \\ tpropagate(\langle M, l[\Gamma]\rangle, S', K, C), \text{ otherwise} \end{cases} \quad (tunit)$$
where
$\Gamma \in K \cup C$,
$\Gamma \equiv l \vee l_1 \vee \ldots \vee l_n$,
$M \not\models l$,
$M \models \neg l_i \wedge \ldots \wedge \neg l_n$
$S' = assert(l, S)$

$$tpropagate(M,S,K,C) := \bot[\Gamma], \text{ where} \quad (tconflict)$$
if $\Gamma \in K \cup C : M \models \neg\Gamma$

$$tpropagate(M,S,K,C) := \langle M, S\rangle, \text{ where} \quad (tterminate)$$
for each $\Gamma \in K \cup C$,
$M \models \Gamma$ or
$\Gamma \equiv l \vee l' \vee \Gamma'$, and $l, l' \notin dom(M)$

Fig. 8. Theory Propagation

7 E-Graph Matching

Most SMT solvers incorporate quantifier reasoning using *matching* over *E-graphs* (i.e., *E-matching*) [Nel81,DNS03]. An E-graph data-structure is the find structure F maintained in Section 4.2. Each equivalence class containing a term t has a canonical representative $F^*(t)$. Let $class(t)$ denotes the equivalence class that contains t, i.e., $\{s \mid F^*(s) = F^*(t)\}$.

Semantically, the formula $\forall x_1, \ldots, x_n.\psi$ is equivalent to the infinite conjunction $\bigwedge_\beta \beta(F)$ where β ranges over all substitutions over the $\overline{x}$. In practice, solvers use heuristics to select from this infinite conjunction those instances that are "relevant" to the conjecture. The key idea is to treat an instance $\beta(\psi)$ as relevant whenever it contains enough terms that are represented in the current E-graph. That is, non-ground terms t_p from ψ are selected as *patterns*, and $\beta(\psi)$ is considered relevant whenever $\beta(t_p)$ is in the E-graph.

An abstract version of the *E-matching* algorithm is shown in Fig. 9. The set of relevant substitutions for a pattern p can be obtained by taking $\bigcup_{t\in E} match(t_p, t, \emptyset)$. The abstract matching procedure returns all substitutions that E-match a pattern t_p with term t. That is, if $\beta \in match(t_p, t, \emptyset)$ then $U \cup \beta \models t_p = t$, and conversely, if $U \cup \beta \models t_p = t$, then there is a β' congruent to β such that $\beta' \in match(t_p, t, \emptyset)$.

$$\begin{aligned} match(x,t,\mathcal{S}) &:= \{\beta \cup \{x \mapsto t\} \mid \beta \in \mathcal{S}, x \notin dom(\beta)\} \cup \\ &\quad \{\beta \mid \beta \in \mathcal{S}, F^*(\beta(x)) = F^*(t)\} \\ match(c,t,\mathcal{S}) &:= \mathcal{S} \text{ if } c \in class(t) \\ match(c,t,\mathcal{S}) &:= \emptyset \text{ if } c \notin class(t) \\ match(f(p_1,\ldots,p_n),t,\mathcal{S}) &= \bigcup_{f(t_1,\ldots,t_n)\in class(t)} match(p_n,t_n,\ldots,match(p_1,t_1,\mathcal{S})) \end{aligned}$$

Fig. 9. E-matching (abstract) algorithm

8 Conclusions

Satisfiability is the process of finding an assignment of values to variables given some constraints on these variables, or explaining why the constraints have no solution. Many computational problems are instances of satisfiability. For this reason, it is important to have efficient solvers for Boolean constraints, constraints over finite domains, constraints in specific theories, and constraints in combinations of theories. SMT solving is an active and exciting area of research with many practical applications. We have presented some of the basic algorithms, but a real implementation requires careful attention to a large number of implementation details and heuristics that we have not covered.

SAT and SMT solving technologies are already making a profound impact on a number of application areas. The theoretical challenges include better representations and algorithms, efficient methods for combining theories and for quantifier reasoning, and various extensions to the basic search method. A lot of experimental work also remains to be done on the careful evaluation of different algorithms and heuristics. In the next few years, satisfiability is likely to become the core engine underlying a wide range of powerful technologies.

References

BDS02. Barrett, C.W., Dill, D.L., Stump, A.: Checking satisfiability of first-order formulas by incremental translation to SAT. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, Springer, Heidelberg (2002)

BTV03. Bachmair, L., Tiwari, A., Vigneron, L.: Abstract congruence closure. Journal of Automated Reasoning 31(2), 129–168 (2003)

CG96. Cherkassky, B.V., Goldberg, A.V.: Negative-cycle detection algorithms. In: European Symposium on Algorithms, pp. 349–363 (1996)

Cra57. Craig, W.: Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. Journal of Symbolic Logic 22(3), 269–285 (1957)

DdM06a. Dutertre, B., de Moura, L.: A fast linear-arithmetic solver for dpll(t). In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 81–94. Springer, Heidelberg (2006)

DdM06b. Dutertre, B., de. Moura, L.: The Yices SMT solver (2006)

DLL62. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem proving. Communications of the ACM 5(7), 394–397 (1962) Reprinted in Siekmann and WrightsonSiekmannWrightson83, pp. 267–270, (1983)

dMRS02. de Moura, L., Rue, H., Sorea, M.: Lazy theorem proving for bounded model checking over infinite domains. In: Voronkov, A. (ed.) Automated Deduction - CADE-18. LNCS (LNAI), vol. 2392, pp. 438–455. Springer, Heidelberg (2002)

dMRS04. de Moura, L., Rue\ss, H., Shankar, N.: Justifying equality. In: Proceedings of PDPAR '04 (2004)

DNS03. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: A theorem prover for program checking. In: Technical Report HPL-2003-148, Hewlett-Packard Systems Research Center (2003)

DP60. Davis, M., Putnam, H.: A computing procedure for quantification theory. JACM 7(3), 201–215 (1960)

ES03. Eén, N., Sörensson, N.: An extensible SAT-solver. In: SAT 2003 (2003)

FJOS03. Flanagan, C., Joshi, R., Ou, X., Saxe, J.B.: Theorem proving using lazy proof explication. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 355–367. Springer, Heidelberg (2003)

GF64. Galler, B.A., Fisher, M.J.: An improved equivalence algorithm. Commun. ACM 7(5), 301–303 (1964)

GN02. Goldberg, E., Novikov, Y.: Berkmin: A fast and robust sat solver (2002)

Koz77. Kozen, D.: Complexity of finitely presented algebras. In: Conference Record of the Ninth Annual ACM Symposium on Theory of Computing, pp. 164–177, Boulder, Colorado (May 2–4 ,1977)

McM05. McMillan, K.L.: An interpolating theorem prover. Theor. Comput. Sci. 345(1), 101–121 (2005)

MMZ+01. Matthew, W., Moskewicz, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an Efficient SAT Solver. In: Proceedings of the 38th Design Automation Conference (DAC'01) (June 2001)

MSS99. Marques-Silva, J., Sakallah, K.: GRASP: A search algorithm for propositional satisfiability. IEEE Transactions on Computers 48(5), 506–521 (1999)

Nel81. Nelson, G.: Techniques for program verification. Technical Report CSL-81-10, Xerox Palo Alto Research Center, Palo Alto, Ca (1981)

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

NO05. Nieuwenhuis, R., Oliveras, A.: Robert Nieuwenhuis and Albert Oliveras. In: Giesl, J. (ed.) RTA 2005. LNCS, vol. 3467, pp. 453–468. Springer, Heidelberg (2005)

NOT06. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). J. ACM 53(6), 937–977 (2006)

Opp80. Derek, C.: Complexity, convexity and combinations of theories. Theoretical Computer Science 12, 291–302 (1980)

Rya04. Ryan, L.: Efficient algorithms for clause-learning SAT solvers. Master's thesis, Simon Fraser University, M.Sc. Thesis (2004)

Sha05. Shankar, N.: Inference systems for logical algorithms. In: Ramanujam, R., Sen, S. (eds.) FSTTCS 2005. LNCS, vol. 3821, pp. 60–78. Springer, Heidelberg (2005)

Sho78. Shostak, R.: An algorithm for reasoning about equality. Comm. ACM 21, 583–585 (1978)

Sho79. Shostak, R.: A practical decision procedure for arithmetic with function symbols. JACM 26(2), 351–360 (1979)

SKC96. Selman, B., Kautz, H., Cohen, B.: Local search strategies for satisfiability testing. In: Johnson, D.S., Trick, M.A. (eds.) Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge, vol. 26 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science. AMS (1996)

SW83. Siekmann, J., Wrightson, G. (eds.): Automation of Reasoning: Classical Papers on Computational Logic, vol. 1 & 2. Springer, Heidelberg (1983)

Tar75. Tarjan, R.E.: Efficiency of a good but not linear set union algorithm. J. ACM 22(2), 215–225 (1975)

WIGG05. Wang, C., Ivančić, F., Ganai, M., Gupta, A.: Deciding separation logic formulae by SAT and incremental negative cycle elimination. In: Sutcliffe, G., Voronkov, A. (eds.) LPAR 2005. LNCS (LNAI), vol. 3835, pp. 322–336. Springer, Heidelberg (2005)

Zha97. Zhang, H.: SATO: An efficient propositional prover. In: Conference on Automated Deduction, pp. 272–275 (1997)

Zha03. Zhang, L.: Searching for Truth: Techniques for Satisfiability of Boolean Formulas. PhD thesis, Princeton University (2003)

ZM02. Zhang, L., Malik, S.: The quest for efficient boolean satisfiability solvers. In: Voronkov, A. (ed.) Proceedings of CADE-19, Berlin, Germany, Springer, Heidelberg (2002)