CAV Tutorial on Satisfiability Modulo Theories July 2007

A Tutorial on Satisfiability Modulo Theories[a]

N. Shankar
(with Leonardo de Moura and Bruno Dutertre)
shankar@csl.sri.com
URL: http://www.csl.sri.com/~shankar/

Computer Science Laboratory
SRI International
Menlo Park, CA

Overview

- Satisfiability is the problem of determining if a formula has a model.
- In the purely *Boolean* case, a model is a truth assignment to the Boolean variables.
- In the *first-order* case, a model assigns values from a domain to variables and interpretations over the domain to the function and predicate symbols.
- For theories such as arithmetic, a model admits a specific (range of) interpretation(s) to the arithmetic symbols.
- Efficient SAT and SMT solvers have many exciting applications.

Goals

This tutorial covers the pragmatic issues in the theory, construction, and use of SMT solvers.

It is not a comprehensive survey, but a basic and rigorous introduction to some of the key ideas.

It is not directed at experts but at potential users and developers of SMT solvers.

We present a series of inference algorithms in a pseudocode form that is

- Abstract enough for theoretical analysis
- But concrete enough to be easily and efficiently implementable.

Historical Background

Satisfiability is one of the central concerns of logic and computation.

It also plays a crucial role in the classification of computational complexity, e.g., NP, #P, and PSPACE.

SMT solvers combining SAT solving and theory solving have been developed since the late 1970s with Nelson and Oppen's Stanford Pascal Verifier and Shostak's STP.

Shostak's STP code still drives PVS.

The Stanford Pascal Verifier begat the influential SMT-based prover Simplify.

SAT Solving

Propositional satisfiability has been actively studied for many decades.

The Davis–Putnam procedure was proposed in 1960 and optimized and implemented in the Davis–Putnam–Logemann–Loveland (DPLL) procedure of 1963.

Modern SAT procedures can routinely handle problems of impressive scale with hundreds of thousands of variables and clauses.

Efficient SAT solving has a number of applications in hardware analysis, bounded and symbolic model checking, and constraint solving.

DPLL-Based SMT Solvers

The extension of DPLL-based SAT solving with theory solving capabilities occurred only recently.

The lazy combination method where the SAT solver updates and queries a theory solvers, was introduced in Verifun, CVC, MathSAT, and ICS.

The eager combination method where a theory solver is used to enumerate lemmas in Boolean form, is best represented by UCLID.

This tutorial focuses on the lazy approach, but the eager approach is also being actively researched.

The basic ideas of SMT solving are fairly simple, but a lot of experimentation and fine-tuning is needed to build an efficient implementation.

Applications and Extensions

SAT and SMT have a large and growing list of applications: Test generation, bounded and symbolic model checking, k-induction and invariance checking, extended static checking, invariant generation, image computation, symbolic model checking, predicate abstraction, scheduling, planning, ...

Extensions to SAT/SMT include proofs, normal forms, unsatisfiable cores, interpolants, MaxSAT/MaxSMT.

Outline

- Logic background (30 min)
- SAT solving (30 min)
- SMT solving (30 min)
- Coffee Break (30 min)
- Theory Constraint Solvers (45 min)
- Theory Combinations (25 min)
- Applications (20 min)

Logic Basics

Logic studies the *trinity* between *language*, *interpretation*, and *proof*.

Language circumscribes the syntax that is used to construct sensible assertions.

Interpretation ascribes an intended sense to these assertions by *fixing* the meaning of certain symbols, e.g., the logical connectives, and *delimiting the variation* in the meanings of other symbols, e.g., variables, functions, and predicates.

An assertion is *valid* if it holds in all interpretations.

Checking validity through interpretations is typically *impossible*, so *proofs* in the form axioms and inference rules are used to demonstrate the validity of assertions.

Language

Signature $\Sigma[X]$ contains functions and predicate symbols with associated arities, and X is a set of variables.

The signature can be used to construct

- *Terms* $\tau := x \mid f(\tau_1, \ldots, \tau_n)$
- *Atoms* $\alpha := p(\tau_1, \ldots \tau_n)$,
- *Literals* $\lambda := \alpha \mid \neg\alpha$
- *Constraints* $\lambda_1 \wedge \ldots \wedge \lambda_n$,
- *Clauses* $\lambda_1 \vee \ldots \vee \lambda_n$,
- *Formulas* $\psi := p(\tau_1, \ldots, \tau_n) \mid \tau_0 = \tau_1 \mid \neg\psi_0 \mid$
  $\psi_0 \vee \psi_1 \mid \psi_0 \wedge \psi_1 \mid (\exists x : \psi_0) \mid (\forall x : \psi_0)$

Structure

A Σ-structure M consists of

- A *domain* $|M|$
- A map $M(f)$ from $|M|^n \to M$ for each n-ary function $f \in \Sigma$
- A map $M(p)$ from $|M|^n \to \{\top, \bot\}$ for each n-ary predicate p.

$\Sigma[X]$-structure M also maps variables in X to domain elements in $|M|$.

E.g., If $\Sigma = \{0, +, <\}$, then M such that $|M| = \{a, b, c\}$ and $M(0) = a$, $M(+) =$
$\{\langle a,a,a\rangle, \langle a,b,b\rangle, \langle a,c,c\rangle, \langle b,a,b\rangle, \langle c,a,c\rangle, \langle b,b,c\rangle, \langle b,c,a\rangle, \langle c,b,a\rangle, \langle c,c,c\rangle\}$,
and $M(<) = \{\langle a,b\rangle, \langle b,c\rangle\}$ is a Σ-structure

Interpreting Terms

$$\begin{aligned} M[\![x]\!] &= M(x) \\ M[\![f(s_1, \ldots, s_n)]\!] &= M(f)(M[\![s_1]\!], \ldots, M[\![s_n]\!]) \end{aligned}$$

Example: From previous example, if $M(x) = a$, $M(y) = b$, and $M(z) = c$, then $M[\![+(+(x, y), z)]\!] = M(+)(M(+)(M(x), M(y)), M(z)) = M(+)(b, c) = a$.

Interpreting Formulas

The interpretation of a formula A in M, $M[\![A]\!]$, is defined as

$$
\begin{aligned}
M \models s = t &\iff M[\![s]\!] = M[\![t]\!] \\
M \models p(s_1, \ldots, s_n) &\iff M(p)(\langle M[\![s_1]\!], \ldots, M[\![s_n]\!] \rangle) = \top \\
M \models \neg\psi &\iff M \not\models \psi \\
M \models \psi_0 \vee \psi_1 &\iff M \models \psi_0 \text{ or } M \models \psi_1 \\
M \models \psi_0 \wedge \psi_1 &\iff M \models \psi_0 \text{ and } M \models \psi_1 \\
M \models (\forall x : \psi) &\iff M\{x \mapsto \mathbf{a}\} \models \psi, \text{ for all } \mathbf{a} \in |M| \\
M \models (\exists x : \psi) &\iff M\{x \mapsto \mathbf{a}\} \models \psi, \text{ for some } \mathbf{a} \in |M|
\end{aligned}
$$

Interpretation Example

$M \models (\forall y : (\exists z : +(y, z) = x))$.

$M \not\models (\forall x : (\exists y : x < y))$.

$M \models (\forall x : (\exists y : +(x, y) = x))$.

Validity

A $\Sigma[X]$-formula A is *satisfiable* if there is a $\Sigma[X]$-interpretation M such that $M \models A$.

Otherwise, the formula A is *unsatisfiable*.

If a formula A is satisfiable, so is its existential closure $\exists \overline{x} : A$, where $\overline{x}$ is $vars(A)$, the set of free variables in A.

If a formula A is unsatisfiable, then the negation of its existential closure $\neg \exists \overline{x} : A$ is *valid*, e.g., $\neg(\forall x : (\exists y : x < y))$.

If $A \wedge \neg B$ is unsatisfiable, $A \Rightarrow B$ is valid.

Propositional Logic

Formulas: $\phi := P \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid \phi_1 \wedge \phi_2$.

P is a class of propositional variables (0-ary predicates):

$p_0, p_1, \ldots$

A model M assigns truth values $\{\top, \bot\}$ to propositional variables: $M(p) = \top \iff M \models p$.

$M[\![\phi]\!]$ is the meaning of ϕ in M and is computed using truth tables:

ϕ	A	B	$\neg A$	$A \vee B$	$A \wedge B$
$M_1(\phi)$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$
$M_2(\phi)$	$\bot$	$\top$	$\top$	$\top$	$\bot$
$M_3(\phi)$	$\top$	$\bot$	$\bot$	$\top$	$\bot$
$M_4(\phi)$	$\top$	$\top$	$\bot$	$\top$	$\top$

Satisfiability in a Theory

A *theory* can be defined in terms of its axioms or as a class of models $\mathcal{T}$ closed under isomorphism and variable reassignment.

A formula A is *satisfiable in a theory* $\mathcal{T}$ ($\mathcal{T} \nsim A$) if for some $M \in \mathcal{T}$, $M \models A$.

Example: The theory of a transitive relation $<$ is set of Σ-structures M where $\Sigma = \{<\}$, and for all $a, b, c \in |M|$, if $M(<)(a, b) = M(<)(b, c) = \top$, then $M(<)(a, c) = \top$.

A first-order theory is the set of models of some set of first-order sentences (the axioms) — hence we do not distinguish between the semantic and axiomatic notions of a theory.

Satisfiability Problems

Satisfiability problems can be classified as follows:

1. Word Problem (WP): $\mathcal{T} \stackrel{?}{\mid\!\sim} \neg p$
2. Uniform WP (UWP): $\mathcal{T} \stackrel{?}{\mid\!\sim} \neg p \wedge p_1 \wedge \ldots \wedge p_n$
3. Clausal Validity (CVP): $\mathcal{T} \stackrel{?}{\mid\!\sim} \Gamma$, for $\Gamma \equiv l_1 \wedge \ldots \wedge l_n$
4. CNF Satisfiability (CNFSP): $\mathcal{T} \stackrel{?}{\mid\!\sim} \Delta_1 \wedge \ldots \wedge \Delta_n$
5. Ground Satisfaction (GSP): $\mathcal{T} \stackrel{?}{\mid\!\sim} A$ for quantifier-free formula A
6. Satisfiability (SP): $\mathcal{T} \stackrel{?}{\mid\!\sim} A$, for any formula A.

Satisfiability Example

Consider the theory of a transitive relation $<$.

The UWP query $\neg x < w, x < y, x < z, y < w$ can be shown to be unsatisfiable by computing the transitive closure of the antecedents to get $x < y, x < z, y < w, x < w$.

The UWP query $\neg y < z, x < y, x < z, y < w$ can be shown to be satisfiable in the term model given by the transitive closure $x < y, x < z, y < w, x < w, y \not< x, z \not< x, y \not< z, z \not< y, z \not< w, w \not< z, w \not< x, w \not< y$.

In the above case, the clausal validity problem is reducible to the UWP: $\neg p_1, \ldots, \neg p_m, q_1, \ldots, q_n$ is unsatisfiable iff $\neg p_i, q_1, \ldots, q_n$ is unsatisfiable for some i, $1 \leq i \leq m$.

Inference Systems

Inference Structures

A $\Sigma[X]$-*inference structure* $\mathcal{I}$ is a triple $\langle \Phi, \vdash, \Lambda \rangle$ of

1. *Logical states* φ

2. *Reduction* relation $\vdash$ between logical states

3. *Logical content* operation Λ, where $\Lambda(\varphi)$ is a $\Sigma[X]$-formula, for each $\varphi \in \Phi$.

$vars(\varphi)$ is the set of free variables in $\Lambda(\varphi)$.

Inference Structures (contd.)

A logical state φ consists of zero or more configurations $\kappa_1 | \ldots | \kappa_n$, where $|$ is associative, commutative, and

$$\Lambda(\kappa_1 | \ldots | \kappa_n) = \Lambda(\kappa_1) \vee \ldots \vee \Lambda(\kappa_n).$$

A state φ is satisfiable if $\Lambda(\varphi)$ is satisfiable.

$\bot$ is an unsatisfiable configuration, so that $\varphi \mid \bot = \varphi$.

The reduction relation must be *monotonic*: If $\varphi \vdash \varphi'$ and $vars(\varphi') \cap vars(\phi) \subseteq vars(\varphi)$, then $\phi|\varphi \vdash \phi|\varphi'$.

Inference Systems

An inference system $\mathcal{I}$ for a Σ-theory $\mathcal{T}$ is a $\Sigma[X]$-inference structure that is

1. **Conservative:** Whenever $\varphi \vdash_{\mathcal{I}} \varphi'$, $\Lambda(\varphi)$ and $\Lambda(\varphi')$ are $\mathcal{T}$-equisatisfiable.

2. **Progressive:** The reduction relation $\vdash_{\mathcal{I}}$ should be well-founded.

3. **Canonizing:** A state is irreducible only if it is either $\bot$ or is $\mathcal{T}$-satisfiable.

For any class of $\Sigma[X]$-formulas Ψ, if there is a mapping ν from Ψ to Φ such that $\Lambda(\nu(A)) = A$, then a $\mathcal{T}$-inference system is a sound and complete decision procedure for $\mathcal{T}$-satisfiability (relative to an oracle for $\vdash$) for Ψ.

Inference System for Equivalence

Delete	$\dfrac{x = y, G; F; D}{G; F; D}$ if $F(x) \equiv F(y)$
Merge	$\dfrac{x = y, G; F; D}{G; F' \circ F; F'(D)}$ if $F(x) \not\equiv F(y)$ $F' = \{orient(F(x) = F(y))\}$
Diseq	$\dfrac{x \neq y, G; F; D}{G; F; x' \neq y', D}$ $x' = F(x), y' = F(y')$
Contrad	$\dfrac{G; F; x \neq y, D}{\bot}$ if $F(x) = F(y)$

Partition F is an idempotent variable equality set $\{x_1 = y_1, \ldots, x_n = y_n\}$, where $x_i \succ y_i$. Note that $F(y) = y$ for $y \notin dom(F)$.

$M[\![a = b]\!] := (M[\![a]\!] = M[\![b]\!])$.

The above inference system is (strongly) conservative, progressive, and canonizing.

Satisfiability using DPLL

Normal Forms

Conversion to normal form is done by an inference system without canonicity.

The conversion steps should converge while preserving satisfiability.

Example normal forms include negation normal form (NNF), conjunctive normal form (CNF), disjunction normal form (DNF), and Skolem normal form.

Such transformations need not actually detect unsatisfiability.

Negation

The inference relation $\vdash$ will be represented by a function f such that the relation $\kappa \vdash f(\kappa)$ is an inference relation.

The decision procedure is then given by $f^*(\kappa_0)$ for the initial state κ_0, or by a tail-recursive function f on the state.

$$\begin{aligned}
\overline{p} &= \neg p \\
\overline{\neg\phi} &= \phi \\
\overline{\phi_0 \vee \phi_1} &= \overline{\phi_0} \wedge \overline{\phi_1} \\
\overline{\phi_0 \wedge \phi_1} &= \overline{\phi_0} \vee \overline{\phi_1}
\end{aligned}$$

Theorem: $\overline{A} \iff \neg A$.

Negation Normal Form

$$
\begin{aligned}
NNF(p) &= p \\
NNF(\neg p) &= \neg p \\
NNF(\neg\neg\phi) &= NNF(\phi) \\
NNF(\phi_0 \vee \phi_1) &= NNF(\phi_0) \vee NNF(\phi_1) \\
NNF(\neg(\phi_0 \vee \phi_1)) &= NNF(\neg\phi_0) \wedge NNF(\neg\phi_1) \\
NNF(\phi_0 \wedge \phi_1) &= NNF(\phi_0) \wedge NNF(\phi_1) \\
NNF(\neg(\phi_0 \wedge \phi_1)) &= NNF(\neg\phi_0) \vee NNF(\neg\phi_1)
\end{aligned}
$$

E.g., $\neg(p \wedge (q \vee \neg r))$ becomes $\neg p \vee (\neg q \wedge r)$.

Theorem: $NNF(A) \iff A$.

Conversion to Clausal (CNF) Form

In *clausal form*, the formula is a set (conjunction) of clauses $\bigwedge K$, and each *clause* Γ in K is a disjunction of literals.

$$
\begin{aligned}
CNF(p, \Delta) &= \langle p, \Delta \rangle \\
CNF(\neg p, \Delta) &= \langle \neg p, \Delta \rangle \\
CNF(\phi_1 \wedge \ldots \wedge \phi_n, \Delta) &= \langle l, \Delta' \rangle, \text{ where } \Delta_0 = \Delta \\
&\quad \langle l_{i+1}, \Delta_i \rangle = CNF(\phi_{i+1}, \Delta_i), \text{ for } 0 \leq i < n \\
&\quad l \equiv p, \Delta' = \Delta_n, \text{ if } p = l_1 \wedge \ldots \wedge l_n \in \Delta_n, \\
&\quad \text{or } l \equiv \neg p, \Delta' = \Delta_n, \text{ if } p = \overline{l_1} \vee \ldots \vee \overline{l_n} \\
&\quad \text{or } l \equiv p, \Delta' = \Delta_n \cup \{p = l_1 \wedge \ldots \wedge l_n\} \\
&\quad \text{for } p \text{ fresh , otherwise} \\
CNF(\phi_1 \vee \ldots \vee \phi_n, \Delta) &= \ldots
\end{aligned}
$$

Resolution

Input K is a set of clauses.

Atoms are ordered by $\succ$ which is lifted to literals so that $\neg p \succ p \succ \neg q \succ q$, if $p \succ q$.

Literals appear in clauses in decreasing order without duplication.

Tautologies, clauses containing both l and $\bar{l}$, are deleted from initial input.

Res	$\dfrac{K, l \vee \Gamma_1, \bar{l} \vee \Gamma_2}{K, l \vee \Gamma_1, \bar{l} \vee \Gamma_2, \Gamma_1 \vee \Gamma_2}$ $\Gamma_1 \vee \Gamma_2 \notin K$ $\Gamma_1 \vee \Gamma_2$ is not tautological
Contrad	$\dfrac{K}{\bot}$ if $p, \neg p \in K$ for some p

Ordered Resolution: Example

$$\dfrac{\dfrac{\dfrac{\dfrac{(K_0 =)\ \neg p \vee \neg q \vee r, \quad \neg p \vee q, \quad p \vee r, \quad \neg r}{(K_1 =)\ \neg q \vee r, \quad K_0}\ \textbf{Res}}{(K_2 =)\ q \vee r, \quad K_1}\ \textbf{Res}}{(K_3 =)\ r, \quad K_2}\ \textbf{Res}}{\bot}\ \textbf{Contrad}$$

Correctness

Progress: Bounded number of clauses in the given literals. Each application of **Res** generates a new clause.

Conservation: For any model M, if $M \models l \vee \Gamma_1$ and $M \models \bar{l} \vee \Gamma_2$, then $M \models \Gamma_1 \vee \Gamma_2$.

Canonicity: Given an irreducible non-$\bot$ configuration K in the atoms $p_1, \ldots, p_n$ with $p_i \prec p_{i+1}$ for $1 \leq i \leq n$, build a series of partial interpretations M_i as follows:

1. Let $M_0 = \emptyset$

2. If p_{i+1} is the maximal literal in a clause $p_{i+1} \vee \Gamma \in K$ and $M_i \not\models \Gamma$, then let $M_{i+1} = M_i\{p_{i+1} \mapsto \top\}$.

   Otherwise, let $M_{i+1} = M_i\{p_{i+1} \mapsto \bot\}$.

Each M_i satisfies all the clauses in K in the atoms $p_1, \ldots, p_i$.

Non-Branching Inference Systems

In *non-branching* systems, the logical state $\kappa_1 \mid \ldots \mid \kappa_n$ consists of exactly one configuration.

E.g., Resolution, equality.

Non-branching systems can still be nondeterministic in the application of inference rules.

Most of the inference systems we present are non-branching.

Semantic Inference Systems

Each configuration κ in a semantic inference system is associated with a partial interpretation M_κ which serves as a candidate model.

When $M_\kappa \models \neg\Lambda(\kappa)$ is detected, there is a *conflict*.

Canonicity can be easily established since the model is part of the state.

M_κ can be used to quickly prune search space by eliminating choices that conflict with the model.

Semantic inference systems operate by incrementally building the candidate model through conflict location, correction, and elaboration.

CNF Satisfiability with DPLL

DPLL is a semantic inference system.

Let K represent the input clause set and $atoms(K)$ represent the propositional variables in K.

The state consists of a quadruple $\langle h, M, K, C \rangle$, where

1. $h \geq 0$ is the branching level of the search
2. M is a sequence $M_0; \ldots; M_h$, where each M_i is a partial map from $atoms(K)$ to $\{\top, \bot\}$ so that $dom(M_i) \cap dom(M_j) = \emptyset$ for $i \neq j$.
3. C is the set of *conflict clauses* derived from K.

M can be viewed as a partial assignment so that $M(l) = M_i[\![l]\!]$ if $l \in dom(M_i)$ or $\bar{l} \in dom(M_i)$ for some i, and $\bot$, otherwise. Then $dom(M) = \{l | l \in M_i \text{ or } \bar{l} \in M_i\}$.

DPLL Informally

If M is a total assignment such that $M \models \Gamma$ for each $\Gamma \in K$, then $M \models K$.

If M is a partial assignment at level h, then *propagation* extends M at level h with the *implied literals* l such that $l \vee \Gamma \in K \cup C$ and $M \models \neg\Gamma$.

If M detects a conflict, i.e., a clause $\Gamma \in K \cup C$ such that $M \models \neg\Gamma$, then the conflict is *analyzed* to construct a conflict clause that allows the search to be continued from a prior level.

If M cannot be extended at level h and no conflict is detected, then an unassigned literal l is *selected* and assigned at level $h + 1$ where the search is continued.

DPLL

Initially, the level is 0, the model M is the empty sequence, and the conflict clause set is empty.

$$\begin{array}{rcll} dpll(K) & := & dpllr(0, \emptyset, K, \emptyset) & (init) \\ dpllr(0, M, K, C) & := & \bot, \text{ if} & (contrad) \\ & & propagate(M, K, C) = \bot[\Gamma] & \end{array}$$

A conflict is when for some $\Gamma \in K \cup C$, $M \models \neg\Gamma$.

A conflict at level 0 signals unsatisfiability since all the literals in M are implied by K.

DPLL

If there is a conflict Γ at level $h+1$, $analyze(h+1, M, \Gamma)$ is used to construct a *conflict clause* Γ' which contains exactly one literal at level $h+1$. Note that $M \models \neg\Gamma'$.

$L2(\Gamma')$ is the highest level of any literal in Γ below $h+1$.

$$
\begin{array}{lll}
dpllr(h+1, M, K, C) & := & dpllr(h', M_{h'} \circ \langle l[\Gamma'] \rangle, K, C'), \\
 & & \text{where} \qquad\qquad\qquad\qquad\qquad\qquad (backjump) \\
 & & propagate(M, K, C) = \bot[\Gamma], \\
 & & analyze(h+1, M, \Gamma) = l[\Gamma'], \\
 & & C' = C \cup \{\Gamma'\}, \\
 & & h' = L2(\Gamma')
\end{array}
$$

DPLL

If $propagate(M, K, C)$ does not detect a conflict, then it returns an assignment M' extending M at level h.

The search proceeds at level $h+1$ by selecting and assigning an unassigned literal l.

$$
\begin{aligned}
dpllr(h, M, K, C) \quad := \quad & dpllr(h+1, M'', K, C), \text{ where } \quad (split) \\
& M' = propagate(M, K, C) \neq \bot, \\
& l = select(M', K) \neq \bot, \\
& M'' = M'; l
\end{aligned}
$$

DPLL

If $propagate(M, K, C)$ does not detect a conflict and returns an assignment M', then if there are no more unassigned literals, $select(M', K)$ returns $\perp$.

The search then returns M' as a satisfying assignment to the input clauses K

$$dpllr(h, M, K, C) \quad := \quad M', \text{ where} \qquad (sat)$$
$$M' = propagate(M, K, C) \neq \perp,$$
$$select(M', K) = \perp$$

Propagation

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

Analysis

Given a conflict Γ such that $M \models \overline{\Gamma}$, resolve on the literals falsified in M_h until there is exactly such literal.

$$
\begin{array}{lcl}
analyze(h, M, \Gamma) & := & l[\Gamma], \\
 & & \text{if there is a unique } l \in \Gamma : M_h \models \bar{l} \\
analyze(h, M, \Gamma) & := & analyze(h, M, \Gamma' \vee \Gamma''), \text{ otherwise} \\
 & & \text{where } \Gamma \equiv l \vee \Gamma', \\
 & & \bar{l}[\bar{l} \vee \Gamma''] \in M_h
\end{array}
$$

Note that the resolution step with $l \vee \Gamma'$ and $\bar{l} \vee \Gamma''$ is feasible since $M \models \overline{\Gamma' \vee \Gamma''}$.

DPLL Example

Let K be

$\{p \vee q, \neg p \vee q, p \vee \neg q, s \vee \neg p \vee q, \neg s \vee p \vee \neg q, \neg p \vee r, \neg q \vee \neg r\}$.

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	$\emptyset$	$\neg p \vee q$

DPLL Example (contd.)

step	h	M	K	C	Γ
conflict	2	$;s;r,\neg q,p$	K	$\emptyset$	$\neg p \vee q$
analyze	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	$\neg q \vee \neg r$

DPLL Correctness

Progress: Each backjump step adds a new assignment at the level h' so that $\sum_{i=0}^{h} |M_i| * (N+1)^{(N-h)}$ increases toward the bound $(N+1)^{(N+1)}$ for $N = |vars(K)|$. In the example, $N = 4$, the backjump step goes from a value 1300 in base 4 to the value 10000 which is closer to the bound 40000.

Conservation: In each transition from $\langle M, K, C\rangle$ to $\langle M', K', C'\rangle$ (or $\bot$), the clause sets $M_0 \cup K \cup C$ and $M_0 \cup K' \cup C'$ are equisatisfiable.

Canonicity: In an irreducible non-$\bot$ state, M is total assignment and there is no conflict so for each clause Γ in $K \cup C$, $M \models \Gamma$.

DPLL with Proof

Resolution proofs can be easily extracted from the analysis phase of the DPLL search procedure.

Each conflict clause in C has an associated proof.

The proof of the final conflict can be derived by resolution from the clauses in $K \cup C$.

Most solvers do not maintain proofs due to the time/space overhead.

A cheaper alternative is to maintain *explanations* in the form of the input clauses used to prove each conflict clause.

Satisfiability Modulo Theories (SMT)

Satisfiability Modulo Theories (SMT)

In SMT solving, the Boolean atoms represent constraints over individual variables ranging over integers, reals, datatypes, and arrays.

The constraints can involve theory operations, equality, and inequality.

Now, the SAT solver has to interact with a theory constraint solver.

The constraint solver can detect conflicts involving theory reasoning, e.g., $f(x) \neq f(y), x = y$, or $x - y \leq 2, y - z \leq -1, z - x \leq -3$.

The constraint solver must produce efficient explanations of unsatisfiability, and support inexpensive queries, incremental assertions, and efficient backtracking.

Example Constraint Solvers

Core theory: Equalities between variables $x = y$, offset equalities $x = y + c$.

Term equality: Congruence closure for uninterpreted function symbols

Difference constraints: Incremental negative cycle detection for inequality constraints of the form $x - y \leq k$.

Linear arithmetic constraints: Fourier's method, Simplex.

Theory Constraint Solver Interface

The satisfiability procedure uses a theory constraint solver oracle which maintains the theory state S with the interface operations:

1. $assert(l, S)$ adds literal l to the theory state S returning a new state S' or $\bot[\Delta]$

2. $check(S)$ checks if the conjunction of literals asserted to S is satisfiable, and returns either $\top$ or $\bot[\Delta]$.

3. $retract(S, l)$: Retracts, in reverse chronological order, the assertions up to and including l from state S.

4. $ask(l, S)$ is an incomplete test the satisfiability of $\bar{l}, S$ returning either $\bot[\Delta]$ or $\top$. A complete test can be implemented as $retract(check(assert(\bar{l}, S)), \bar{l})$.

Generic SMT

The $tdpll$ procedure operates on a state consisting of

1. Level h
2. Partial assignment M
3. Theory solver state S
4. Input clause set K and the conflict clause set C.

Initially, the level is 0 and M, S, and C are empty.

$$tdpll(K) \quad := \quad tdpllr(0, \emptyset, \emptyset, K, \emptyset) \quad (tinit)$$

Generic SMT

In each decision level, the $ask(l, S)$ operation is used to scan the unassigned literals l in K to check if $\bar{l}$ is entailed by S.

Following scanning, propagation is used as in $dpll$.

At level 0, if the scan-and-propagate operation $scanprop$ returns a conflict with conflict clause Γ, then the input clause set is unsatisfiable.

$$tdpllr(0, M, S, K, C) \quad := \quad \bot, \text{ where} \qquad (tcontrad)$$
$$scanprop(M, S, K, C) = \bot[\Gamma]$$

Theory Backjump

$$
\begin{aligned}
backjump(h, M, S, K, C, \Gamma) \quad = \quad & \langle h', M_{\overline{h'}} \circ l[\Gamma'], S', C' \rangle, \text{ where} \\
& l[\Gamma'] = tanalyze(h, M, \Gamma) \\
& h' = L2(\Gamma') \\
& C' = C \cup \{\Gamma'\} \\
& S' = retract(S, l_{h'+1})
\end{aligned}
$$

$l_{h'+1}$ is the decision literal at $h' + 1$.

Generic SMT

Backjumping is as in $dpll$ except that S is also backed up to the checkpointed state $S_{h'}$.

$tanalyze$ is also identical to $analyze$ from $dpll$.

$$tdpllr(h+1, M, S, K, C) \quad := \quad tdpllr(h', M', S', K, C'), \qquad (tbackjump)$$

where

$$\langle h', M', S', C' \rangle =$$

$$backjump(h+1, M, S, K, C, \Gamma)$$

$$scanprop(M, S, K, C) = \bot[\Gamma]$$

Generic SMT

The $split$ operation uses $tselect$ to select an unassigned literal and adds it to the the theory state.

The $tselect$ operation is the same as $select$.

$$
\begin{aligned}
&tdpllr(h, M, S, K, C) \\
&\quad := \quad tdpllr(h+1, M'', S'', K, C), \text{ where} \qquad (tsplit) \\
&\qquad\quad \langle M', S', C' \rangle = scanprop(M, S, K, C) \neq \bot, \\
&\qquad\quad l = tselect(M', K) \neq \bot, \\
&\qquad\quad S'' = assert(l, S') \neq \bot \\
&\qquad\quad M'' = M'; l[]
\end{aligned}
$$

Unsatisfiable assert

$$
\begin{aligned}
&tdpllr(h, M, S, K, C) \\
&\quad := \quad tdpllr(h', M', S', K, C'), \text{ where} \qquad\qquad (tsplit) \\
&\qquad\quad backjump(h+1, M', S', K, C, \Gamma) \\
&\qquad\quad \langle M', S', C' \rangle = scanprop(M, S, K, C) \neq \bot, \\
&\qquad\quad l = tselect(M', K) \neq \bot, \\
&\qquad\quad assert(l, S') = \bot[\Gamma]
\end{aligned}
$$

Generic SMT

If the assignment M is total, then either $check(S')$ returns $\bot$ and we have satisfiability, or backjumping is executed.

$$tdpllr(h, M, S, K, C) \qquad (tcheck)$$

$$:= \begin{cases} M', \text{ if } check(S') = \top \\ \bot, \text{ if } h = 0, check(S') = \bot[\Gamma] \\ backjump(h, M', S', K, C, \Gamma), \\ \quad \text{if } h > 0, check(S') = \bot[\Gamma] \end{cases}$$

with

$\langle M', S' \rangle = scanprop(M, S, K, C) \neq \bot,$

$tselect(M', K) = \bot$

Propagation

The scan-and-propagate phase first executes $ask(\bar{l}, S)$ for each unassigned literal l and then performs Boolean propagation.

If $ask(\bar{l}, S) = \bot[\Gamma]$, then l is appended to M, and Γ is appended to C.

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

Propagation

Propagation is similar to $dpll$. Each propagated literal is also asserted to S.

$$tpropagate(M, S, K, C)$$

$$:= \begin{cases} \bot[\Gamma], \text{ if } S' = \bot[\Gamma] \\ tpropagate(\langle M, l[\Gamma]\rangle, S', K, C), \text{ otherwise} \end{cases} \quad (tunit)$$

where

$\Gamma \in K \cup C$,

$\Gamma \equiv l \vee l_1 \vee \ldots \vee l_n$,

$l \notin dom(M)$,

$M \models \overline{l_1} \wedge \ldots \wedge \overline{l_n}$

$S' = assert(l, S)$

Propagation

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

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

TDPLL example

Input is $y = z, \quad x = y \vee x = z, \quad x \neq y \vee x \neq z$

Step	M	F	D	C
Propagate	$y = z$	$\{y \mapsto z\}$	$\emptyset$	$\emptyset$
Select	$y = z; x \neq y$	$\{y \mapsto z\}$	$\{x \neq y\}$	$\emptyset$
Scan	$\ldots, x \neq z$ $[x \neq z \vee y \neq z \vee x = y]$	$\{y \mapsto z\}$	$\{x \neq y\}$	$\emptyset$
Propagate	$\ldots$	$\{y \mapsto z\}$	$\{x \neq y\}$	
Analyze	$\ldots$	$\{y \mapsto z\}$	$\{x \neq y\}$	$\{y \neq z \vee x = y\}$
Backjump	$y = z, x = y$	$\{y \mapsto z\}$	$\{x \neq y\}$	$\{y \neq z \vee x = y\}$
Assert	$y = z, x = y$	$\{x \mapsto y, y \mapsto z\}$	$\{x \neq y\}$	$\{y \neq z \vee x = y\}$
Scan	$\ldots, x = z$ $[x = z \vee x \neq y \vee y \neq z]$	$\{x \mapsto y, y \mapsto z\}$	$\{x \neq y\}$	$\{y \neq z \vee x = y\}$
Conflict				

Theory Constraint Solvers

Theory Constraint Solvers

Theory constraint solvers solve the clausal validity problem in a given theory by checking the satisfiability of $l_1 \wedge \ldots \wedge l_n$.

We first present solvers for individual theories as inference systems.

These solvers implement the $assert$, ask, $check$, $retract$ interfaces.

We then present the combination of individual solvers into a solver for the union of theories.

Variable Equality: Union

The variable equality inference system is similar to the one presented earlier.

The state consists of a *find* structure F, the E-graph, that maintains equivalence classes and the input disequalities D.

With respect to F, we write $x \sim y$ if $F^*(x) = F^*(y)$.

Initially, $F(x) = x$ for each variable x.

The equality $x = y$ is processed by merging distinct equivalence classes using the *union* operation below.

$$union(F)(x,y) \quad = \quad \begin{cases} F[x' := y'], y' \prec x' \\ F[y' := x'], \text{ otherwise} \end{cases}$$

where $x' \equiv F^*(x) \not\equiv F^*(y) \equiv y'$

Merging Input Equalities

$$addeqlit(x = y, F, D) \qquad (skip)$$

$$:= \quad \langle F, D \rangle, \text{ if}$$

$$F^*(x) \equiv F^*(y)$$

$$addeqlit(x = y, F, D) \qquad (union)$$

$$:= \quad \begin{cases} \bot, \text{ if} \\ \quad F'^*(u) \equiv F'^*(v) \text{ for some } u \neq v \in D \\ \langle F', D \rangle, \text{otherwise} \end{cases}$$

where

$$x' = F^*(x) \not\equiv F^*(y) = y',$$

$$F' = union(F)(x, y)$$

Adding Disequalities

$$
\begin{array}{lcll}
addeqlit(x \neq y, F, D) & := & \bot, \text{ if } F^*(x) \equiv F^*(y) & (contrad) \\
addeqlit(x \neq y, F, D) & := & \langle F, D \rangle, \text{ if} & (skipdiseq) \\
 & & F^*(x) \equiv F^*(x'), & \\
 & & F^*(y) \equiv F^*(y'), & \\
 & & \text{for } x' \neq y' \in D & \\
addeqlit(x \neq y, F, D) & := & \langle F, \{x \neq y\} \cup D \rangle, \text{ otherwise.} & (adddiseq)
\end{array}
$$

Correctness

Progress: The *find* trees are *rooted* so that for any x, $F(F^i(x)) = F^i(x)$ for some i.

This means the $F^*(x)$ operation is always well defined and terminating.

Conservation: In $addeqlit(l, F, D) = \langle F', D' \rangle$, the two sides are equisatisfiable, as is also the case when $addeqlit(l, F, D) = \bot$.

Canonicity: When $addeqlit(l, F, D) = \langle F', D' \rangle$, the state $\langle F', D' \rangle$ can be used to construct a term model M with $|M| = \{x \mid F'(x) = x\}$ and $M(x) = F'^*(x)$.

The API

What are ask, $assert$, $check$, and $retract$?

The state S is $\langle F, D \rangle$.

$assert(l, S)$ is just $addeqlit(l, F, D)$.

Since $addeqlit$ is complete, $check$ is just the identity.

$ask(l, S)$ is defined so that

$$ask(x = y, \langle F, D \rangle) = \begin{cases} \bot, \text{ if } F^*(x) = F^*(x'), F^*(y) = F^*(y'), \\ \quad \text{for some } x' \neq y' \in D \\ \top, \text{ otherwise} \end{cases}$$

$$ask(x \neq y, \langle F, D \rangle) = \begin{cases} \bot, \text{ if } F^*(x) = F^*(y) \\ \top, \text{ otherwise} \end{cases}$$

Retracting Assertions

Checkpointing the *find* data structure can be expensive.

Shostak's method for checkpointing uses association stacks so that checkpointing can be done at zero cost by maintaining stack pointers.

However, this makes the *find* operation expensive. This can be ameliorated with a difference alist, but then backtracking becomes expensive.

A disequality can be retracted by just deleting it from D.

Retracting equality assertions is more difficult — the history of the merge operations have to be maintained and demerged.

The Simplify prover employs a space-efficient solution by maintaining the equivalence classes in a circular list.

Congruence Closure

The free theory $\Phi(\Sigma)$ over a signature Σ is the first-order theory with an empty set of non-logical axioms.

Σ-terms have their function symbols from the signature Σ.

Equivalence is extended to *congruence* 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 $1 \leq i \leq n$.

For each term $f(s_1, \ldots, s_n)$, the E-graph now contains nodes v where $label(v)$ is either a variable name or an n-ary function f with $children(v) = v_1, \ldots, v_n$.

The graph is congruence-closed if whenever $label(v) = label(v')$ and $v_i \sim v'_i$ for $1 \leq i \leq n$ for $children(v) = v_1, \ldots, v_n$ and $children(v') = v'_1, \ldots, v'_n$, then $v \sim v'$.

Adding Equality

As with equivalence, the *find* roots $s' = F^*(s)$ and $t' = F^*(t)$ are merged.

Any congruent pairs from the parent sets $\pi(s')$ and $\pi(t')$ must also be merged.

Any pair of terms $\hat{s}$ in $\pi(s')$ and $\hat{t}$ in $\pi(t')$ that are congruent in F are added to the queue of equalities to be merged.

$$addeqlit(s = t, F, D, \pi) \quad := \quad close(F, D, \{s = t\}, \pi)$$

Closing the E-Graph

$$close(F, D, Q, \pi) \qquad (congruence)$$

$$:= \quad close(F', D, Q', \pi'),$$

when $s, t : s = t \in Q, s' = F^*(s) \not\equiv F^*(t) = t',$

$s' \prec t', F' = F[t' := s'],$

$\pi' = \pi[s' := \pi(s) \cup \pi(t')]$

$$Q' = Q \cup \left\{ \hat{s}' = \hat{t}' \;\middle|\; \begin{array}{l} \hat{s} \in \pi(s), \hat{t} \in \pi(t), \\ \hat{s}' = F'^*(\hat{s}) \not\equiv F'^*(\hat{t}) = \hat{t}' \\ congruent(F', s', t') \end{array} \right\}$$

$$close(F, D, Q, \pi) \qquad (terminate)$$

$$:= \quad \langle F, D, \pi \rangle, \text{ otherwise.}$$

Congruence Closure Example

$$x = g(x), f(x, g(g(x))) \neq g(x), f(x, x) = g(g(x))$$

Term universe $U = \{x, g(x), f(x, x), g(g(x)), f(x, g(g(x)))\}$.

Step	F	D
Add	$\{x \mapsto g(x)\}$	$\emptyset$
Close	$\{\ldots, g(x) \mapsto g(g(x)), f(x, x) \mapsto f(x, g(g(x)))\}$	$\emptyset$
Add	$\{\ldots\}$	$f(x, g(g(x))) \neq g(x)$
Add	$\{\ldots, f(x, x) \mapsto g(g(x))\}$	$f(x, g(g(x))) \neq g(x)$
Conflict		

Congruence Closure Correctness

Progress: If there are n distinct equivalence classes, then for each invocation of $close$, at most $(n-1)*(n-2)$ pairs of distinct equivalence classes can be added to Q.

Conservation: In each step of $close$, if $\langle F, D, Q\rangle \vdash \bot$, then $\langle F, D, Q\rangle$ is unsatisfiable. If $\langle F, D, Q\rangle \vdash \langle F', D', Q'\rangle$, then equisatisfiability is easily checked.

Canonicity: The state $\langle F, D\rangle$ yields a term model M where $|M| = \{s \mid F(s) = s\}$, where $M(x) = F^*(x)$ and $M(f)(a_1, \ldots, a_n) = F^*(f(b_1, \ldots, b_n))$ if there is a term $f(b_1, \ldots, b_n) \in |M|$ such that $F^*(a_i) \equiv F^*(b_i)$. Otherwise, $M(f)(a_1, \ldots, a_n) = f(a_1, \ldots, a_n)$.

Difference Arithmetic

Difference arithmetic constraints contain literals of the form $x - y \leq c$ for integer constant c.

Strict inequalities $x - y < c$ can be encoded as $x - y \leq c - 1$.

Such constraints are common in program verification and scheduling applications.

DA constraints can be represented as a weighted directed graph with variables for vertices and an edge from y to x labeled c corresponding to $x - y \leq c$.

An unsatisfiable constraint corresponds to the existence of a path $x_1\langle c_1\rangle x_2 \ldots x_n\langle c_n\rangle x_1$ such that $c_1 + c_2 \ldots + c_n < 0$ since this would imply $x_1 < x_1$.

Difference Arithmetic

The semantic inference system for difference arithmetic below maintains a graph structure E such that $E(y) = \{\langle x_1, c_1\rangle, \ldots, \langle x_n, c_n\rangle\}$ for the constraints $x_i - y \leq c_i$ and an assignment ρ of integers to variables.

An inequality $x - y \leq c$ is asserted by adding a new edge to the graph, and modifying ρ as needed so that $\rho(x) - \rho(y) \leq c$. The new assignment might not be consistent with the edges out of x.

$$addineq(x, y, c, \rho, E) \quad := \quad \langle \rho, E[y := E(y) \cup \{\langle x, c\rangle\}]\rangle, \text{ if } \rho(y) - \rho(y) \leq c$$

Difference Arithmetic

The vertex x is added to the relaxation queue.

If the original vertex y appears in the relaxation queue, then we have detected a negative cycle.

$$addineq(x, y, c, \rho, E) := \begin{cases} \bot, \text{ if } \rho' = \bot \\ \langle \rho', E' \rangle, \text{ otherwise} \end{cases}$$

where

$$E' = E[y := E(y) \cup \{\langle x, c \rangle\}],$$

$$\rho' = \begin{cases} \rho, \text{ if } \rho(x) - \rho(y) > c \\ relaxv(y, \rho[x := \rho(y) + c], E', \{x\}), \\ \text{otherwise} \end{cases}$$

Relaxing

For a selected vertex x in Q, collect all the $\langle z, c\rangle \in E(x)$ such that $\rho(z) - \rho(x) > c$.

Set the assignment of $\rho(z)$ to $\rho(x) + c$, and add z to the queue Q.

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

Difference Arithmetic Correctness

In $relaxv$, the only unsatisfied constraints w.r.t. ρ are those corresponding to edges in $E(x)$ for $x \in Q$.

If the $addineq$ procedure returns an assignment ρ', it is a valid assignment since Q is empty.

If ρ_0 is the input assignment to $relaxv$, then for each vertex $x \in Q$, there is a path θ from y to x such that $W(\theta) = \rho(x) - \rho_0(y) < \rho_0(x) - \rho_0(y)$.

In particular, if $y \in Q$, then there is a cycle θ such that $W(\theta) = \rho(x) - \rho_0(y) < 0$.

Linear Arithmetic

Linear arithmetic constraints have the form $s \leq t$ for linear polynomials s and t over the reals.

These inequalities can be placed in a normal 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

Some of the approaches to solving linear arithmetic constraints include

1. Fourier's method
2. Shostak's loop residue method
3. Simplex: Solve $AX = B$ subject to $X \geq 0$.
4. Simplex in General Form: Solve $AX = B$ subject to $L \leq X \leq U$.

General Form Simplex

Here the solution set has only unrestricted variables with lower and upper bounds.

The state consists of

1. A tableau T of the form $\vec{y} = A\vec{x}$ for basic variables $y_1, \ldots, y_m$ and non-basic variables $x_1, \ldots, x_n$

2. An assignment β for the non-basic variables. The assignment for a basic variable y is $\beta[\![T(y)]\!]$.

3. Two maps, L for the lower bound, and U for the upper bound, from $vars(T)$ to the rationals.

An input constraint of the form $k_1 * x_1 + \ldots + k_n * x_n \leq c$ is converted to $y \leq c$ where $y = k_1 * x_1 + \ldots + k_n * x_n$.

General Form Simplex Solver

A new inequality is added as $z \leq c$, where $z = k_1 * x_1 + \ldots + k_n * x_n$ is already introduced in the initial construction of T.

Now, $L'(z) = \max(L(z), c)$.

Similarly, if the new inequality is $z \geq c$, then $U'(z) = \min(U(z), c)$.

If $L'(z) > U'(z)$, we have an unsatisfiable state.

If $\beta(z) < L'(z)$, then if z is non-basic, we set $\beta' = \beta[z := L'(z)]$.

If for some basic y, $\beta'[\![T(y)]\!] < L'(y)$, then $T' = pivot(T)(x_i, y)$, where $T(y) = b_1 x_1 + \ldots + b_n x_n$, $b_i > 0$ and $\beta(x_i) < u_i$, or $b_i < 0$ and $\beta(x_i) > l_i$.

Simplex Example

$T_0 = \begin{array}{l} s_1 = -x + y \\ s_2 = x + y \end{array}$		$\beta_0 = (x \mapsto 0, y \mapsto 0, s_1 \mapsto 0, s_2 \mapsto 0)$
$T_1 = T_0$	$x \leq -4$	$\beta_1 = (x \mapsto -4, y \mapsto 0, s_1 \mapsto 4, s_2 \mapsto -4)$
$T_2 = T_1$	$-8 \leq x \leq -4$	$\beta_2 = \beta_1$
$T_3 = \begin{array}{l} y = x + s_1 \\ s_2 = 2x + s_1 \end{array}$	$\begin{array}{l} -8 \leq x \leq -4 \\ s_1 \leq 1 \end{array}$	$\beta_3 = (x \mapsto -4, y \mapsto -3, s_1 \mapsto 1, s_2 \mapsto -7)$

Datatypes

The list datatype has the axioms

1. $car(cons(x, y)) = x$

2. $cdr(cons(x, y)) = y$

This implies, for instance that if $cons(x, y) = cons(y, v)$, then $x = y$ and $u = v$.

To deduce this from the axioms, we need to introduce the terms $car(cons(x, y))$ and $cdr(cons(x, y))$ that might both be in the term universe U.

Term Extension

One option is to treat car, cdr, and $cons$ as uninterpreted but introduce instances of the axioms that extend the term universe U.

This can be done lazily based on the contents of the E-graph.

For example, whenever $cons(x, y)$ occurs in U, add the clauses $car(cons(x, y)) = x$, and $cdr(cons(x, y)) = y$.

Yices uses term extension for arrays as well:

1. If $store(f, i, v) \in U$ add $sel(store(f, i, v), i) = v$.

2. For any g such that $g \sim store(f, i, v)$ or $g \sim f$, and $sel(g, j) \in U$, add $i = j \vee sel(store(f, i, v), j) = f(j)$.

E-Graph Closure

Another option is to derive enough closure rules to saturate the E-graph so that no term extension is needed.

Closure rules are derived from the axioms by finding a context $C[]$ such that for $\vec{u} \sim \vec{v}$ and axiom instances $C[\vec{u}] = u'$ such that

1. If $C[\vec{v}] \in U$ and $u' \in U$, merge $C[\vec{v}]$ and u'.
2. If $C[\vec{v}] = v'$ is an axiom instance, $u', v' \in U$, then merge u' and v'.

For datatypes, we get the closure rules:

1. If $cons(s,t) \sim cons(u,v)$, merge s and u, and t and v.
2. If $car(u) \in U$, $u \sim cons(s,t)$, merge $car(u)$ and s.
3. If $cdr(u) \in U$ and $u \sim cons(s,t)$, then merge $cdr(u)$ and t.

Arrays

For arrays without extensionality, the closure rules are

1. If $f \sim store(g, i, v)$ and $j \sim i$ then merge $sel(f, j)$ and v.

2. If $f \sim store(g, i, v)$, $j \sim j'$, $i \sim i'$, $i' \neq j'$, then merge $sel(f, j)$ and v.

3. If $f \sim g$, $i \neq j$, then merge $sel(store(f, i, v), j)$ and $sel(g, j)$.

With extensionality, we need to introduce a new constant so that from $f \neq g$, we add $f(k) \neq g(k)$ for a new constant k.

Combining Theory Solvers

Theory Combination

Practical satisfiability problems involve multiple theories: arithmetic, arrays, datatypes, bit-vectors, and uninterpreted function symbols.

Nelson and Oppen give an elegant method for combining theory solvers:

1. The overall state of the combined solvers consists of a core E-graph S_0 and the individual theory states: $S_1; \ldots; S_m$.

2. To add a mixed literal l, purify it into individual literals of the form $l', x_1 = t_1, \ldots, x_n = t_n$, where the literal l' is in the core and each t_i is a pure term in a theory.

3. Add each literal to the appropriate theory to obtain S'_i.

4. If there is an arrangement A of shared variables into equivalence classes such that $A \cup S'_0 \cup S'_i$ is satisfiable for each theory i, then the literal l is satisfiable with respect to S.

Purification

A quantifier-free Σ-constraint Δ can be *purified* so that each literal in the formula is a Σ_i-literal for $i = 1, 2$.

Let $\Delta[t := s]$ 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.

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

Guessing an Arrangement

A *partition* Π on a set of variables γ is a disjoint collection subsets $\gamma_1, \ldots, \gamma_n$ such that $\bigcup_{i=1}^{n} \gamma_i = \gamma$.

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 $\{x \neq y \mid \text{ for some } i, j : i \neq j, x \in \gamma_i, y \in \gamma_j\}$.

Combined Theory Solver

The state now consists of the core state S_0 and the theory states S_1 and S_2.

The corresponding signatures are $\Sigma_0 = \emptyset$, Σ_1, and Σ_2.

The $addlit(l, S)$ operation can be implemented by

$$
\begin{array}{rcl}
addlit(l, S) & = & addpurelits(R, S) \\
 & & \text{where } R = purify(l, \emptyset) \\
addpurelits(\{l\} \uplus R, S) & = & addpurelits(R, S'), \\
 & & \text{where } S' = S'_0, S'_1, S'_2, \\
 & & S'_i = \begin{cases} addlit_i(l, S_0), & \text{for } l \in \Sigma_i \\ S_i, & \text{otherwise} \end{cases}
\end{array}
$$

Combined Theory Solver

The combined $check$ operation can also be described by guessing an arrangement and checking in the individual theories.

$$check(S_0; S_1; S_2) \quad = \quad \bigvee_{\Pi} check_1(S_0', S_1') \wedge check_2(S_0', S_2')$$

where $S_0', S_1', S_0' = addpurelits(A_\Pi, S_0; S_1; S_2)$

Correctness

The Nelson–Oppen combination works as long as the component theories are stably infinite: If a constraint Δ has a $\mathcal{T}_i$-model, then it has a countable such model.

The union of two consistent stably infinite theories is consistent and stably infinite.

Progress: Each step: purification, guessing, and individual theory solving, converges.

Conservation: A $(\mathcal{T}_1 + \mathcal{T}_2)$-model is a model that is both a $\mathcal{T}_1$-model and a $\mathcal{T}_2$-model. Purification and guessing easily preserve $(\mathcal{T}_1 + \mathcal{T}_2)$-models. Each such model must satisfy one of the arrangements.

If theory solving indicates unsatisfiability, the premise state is also unsatisfiable.

Correctness: Canonicity

If the theory solving stage indicates satisfiability, then by the canonicity of the theory solver, we have countable models M_i of $S_0; S_i$ for $i = 1, 2$.

A combined model M can be constructed. Assume that $M_1(x) = M_2(x)$ for $x \in S_0$.

There is a bijection h with $h(M_1(x)) = M_1(x)$ for $x \in S_0$ such that

$$\begin{aligned} M(f)(a_1, \ldots, a_n) &= h^{-1}(M_2(f)(h(a_1), \ldots, h(a_n)), \text{ for } f \in \Sigma_2 \\ M(p)(a_1, \ldots, a_n) &= M_2(p)(h(a_1), \ldots, h(a_n)), \text{ for } p \in \Sigma_2 \end{aligned}$$

Justification by Interpolation

For satisfiability in countable models, the empty theory admits quantifier elimination.

By interpolation, if $S_0; S_1; S_2$ is unsatisfiable, there is an interpolant I in $\Sigma_1[X_1] \cap \Sigma_2[X_2]$ such that $\mathcal{T}_1 \models S_1 \Rightarrow I$ and $\mathcal{T}_2 \models S_1 \Rightarrow \neg I$.

Since $\Sigma_1 \cap \Sigma_2 = \emptyset$, the interpolant I is a first-order formula in the empty signature.

Stable-infiniteness is not a problem in practice: theories with finite cardinalities, like fixed-width bit-vectors, can be guarded with predicates, e.g,, $bitvector?(x)$.

Quantifier Elimination

A subformula of the form $\exists x : x = y \wedge l_1 \wedge \ldots \wedge l_n$ can be replaced by $l_1[y/x] \wedge \ldots \wedge l_n[y/x]$.

For a subformula of the form $\exists x : l_1 \wedge \ldots \wedge l_n$ where no l_i is of the form $x = y$, the literals containing x can be dropped along with the quantifier, for the case of countable models.

The last step needs stable-infiniteness: for a finite model M where $|M| = \{a, b\}$, with $M(y) = a$ and $M(z) = b$, and a formula A of the form$(\exists x : x \neq y \wedge x \neq z \wedge y \neq z)$, we have $M \not\models A$, but $M \models y \neq z$.

Interpolation

If there is a quantifier-free formula I such that $\mathcal{T}_1 \models S_1 \Rightarrow I$ and $\mathcal{T}_2 \models S_2 \Rightarrow \neg I$.

Let $A_1 \vee \ldots \vee A_n$ be the disjunction of all possible arrangements.

Clearly, I is equivalent to the disjunction of some subset O of the arrangements $\bigvee_{i \in O} A_i$ and $\neg I$ is equivalent to $\bigvee_{i \notin O} A_i$.

If each A_i is unsatisfiable with either S_1 or S_2, then there is an interpolant: $I = \bigvee \{A_i \mid \mathcal{T}_1 \models S_1, A_i\}$.

If some A_i is satisfiable with both S_1 and S_2, then there is no interpolant since each A_i is either part of I or $\neg I$.

Convex Theories

A theory is convex if whenever a set of atoms Δ is such that $\Delta \Rightarrow x_1 = y_1 \vee \ldots x_n = y_n$, then $\Delta \Rightarrow x_i = y_i$ for some i, $1 \leq i \leq n$.

A theory is compact if a set of formulas S is satisfiable iff each finite subset of S is satisfiable.

If a formula in a compact theory has no infinite models, then it has models of cardinality at most m for some natural number m.

Any compact, convex theory with nontrivial models is stably infinite.

Otherwise, there is a constraint Δ with a finite model M of cardinality m so that the formula B_m given by $\bigvee_{i=1}^{m+1} \bigvee_{j=i+1}^{m+1} x_i = x_j$ is satisfied.

If we split Δ into the atoms Δ^+ and negated atoms Δ^-, then $\mathcal{T} \models \Delta^+ \supset \overline{\Delta^-} \vee B_m$ so that $vars(\Delta) \cap vars(B_m) = \emptyset$, but $\mathcal{T} \not\models l$ for any $l \in \overline{\Delta^-} \vee C_m$.

Convex Theories: Examples

Equational theories and Horn theories are examples of convex theories.

Any theory that is closed under direct products (i.e., whenever M_1 and M_2 are $\mathcal{T}$-models, so is $M_1 \times M_2$) is convex.

Non-convex, stably infinite theories include integers ($x - y = 1, x \leq z, z \leq y$), and nonlinear arithmetic ($x^2 = 1, y = 1, z = -1$), and arrays.

For convex theories, there is no need to guess an arrangement.

The arrangement consists of those equalities $x = y$ on shared variables x, y that are implied by each S_i.

Combination of Easy Theories can be Hard

The free Σ-theory $\Phi(\Sigma)$ is decidable in $O(n \log(n))$ by congruence closure.

Difference arithmetic is decidable in $O(mn)$ by the Bellman-Ford algorithm.

The combination is NP-hard: $x_1 \vee \neg x_2 \vee x_3$ can be represented as $f(x_1, x_2, x_3) \neq f(0, 1, 0)$, for $0 \leq x_1, x_2, x_3 \leq 1$.

Satisfiability Modulo Theories Revisited

The guessing of the arrangement can be incorporated into the basic DPLL search procedure.

We need to split on equalities that might not be part of the input.

For example, the array theory requires splitting on $i = j$ whenever we have $store(f, i, v)$ and $sel(g, j)$.

E-Graph Matching

We have only looked at ground satisfiability with built-in decidable theories.

First-order satisfiability is undecidable in general, though there are some very useful decidable fragments, e.g., $\exists\forall$ for any theory where ground satisfiability is decidable.

Quantifiers are convenient for capturing certain rules that are not captured by theories, e.g., $\forall x, y : f(x, y) = f(y, x)$, or $(\forall x : f(g(x)) = g(f(x))$.

Such quantified formulas can be reduced to the ground case by instantiation through matching.

However matching cannot be syntactic — the term universe might contain terms $f(s)$ and $g(t)$ where $s \sim g(t)$, but no syntactic instance of a pattern $f(g(t))$

E-Graph Matching

Given a quantified formula $\forall x_1, \ldots, x_n : A$, a pattern is a term that contains all the quantified variables.

Given a query $t_p \stackrel{?}{=} t$, an abstract E-graph matcher returns a set of matches $\mathcal{S}$ such that for each substitution $\beta \in S$, $E \cup \beta \models t_p = t$, and if for some substitution β, $E \cup \beta \models t_p = t$, then $\beta \in \mathcal{S}$.

$$
\begin{aligned}
match(x, t, \mathcal{S}) &:= \{\beta \cup \{x \mapsto t\} \mid \beta \in \mathcal{S}, x \notin dom(\beta)\} \cup \\
&\quad \{\beta \mid \beta \in \mathcal{S}, F^*(\beta(x)) = F^*(t)\} \\
match(c, t, \mathcal{S}) &:= \mathcal{S} \ \textit{if} \ c \in class(t) \\
match(c, t, \mathcal{S}) &:= \emptyset \ \textit{if} \ c \notin class(t)
\end{aligned}
$$

$$
\begin{aligned}
&match(f(p_1, \ldots, p_n), t, \mathcal{S}) \\
&\quad := \bigcup_{f(t_1, \ldots, t_n) \in class(t)} match \begin{pmatrix} p_n, t_n, \ldots, \\ match(p_1, t_1, \mathcal{S}) \end{pmatrix}
\end{aligned}
$$

Applications

Transition Systems

Given *state* Σ, *initial state predicate* $I(s)$, *next-state relation* $N(s, s')$, and *assertion* $P(s)$.

Bounded Model Checking:
$satisfiable(I(s_0) \wedge \bigwedge_{i=0}^{k-1} N(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{k} \neg P(s_i))$

k-Induction:
$satisfiable(\bigwedge_{i=0}^{k} N(s_i, s_{i+1}) \wedge \bigwedge_{i=0}^{k} P(s_i) \wedge \neg P(s_{k+1}))$

Image computation: Compute the formula representing $\mathbf{AX}_N P$, or $\forall s' : N(s0, s') \wedge P(s')$

Fixpoints: Compute the formula representing $\mathbf{AG}_N P$.

Interpolant: Find interpolant formula F such that
$I(s_0) \wedge N(s_0, s_1) \Rightarrow F(s_1)$ and
$\neg(F(s_1) \wedge \bigwedge_{i=1}^{k-1} N(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{k} P(s_i))$.

Scheduling

Given j jobs and m machines, each job consists of a sequence of tasks $t_{i1}, \ldots, t_{in}$, where each task t_{ik} is a pair $\langle M, \delta \rangle$ for machine M and duration δ.

Find a schedule with a minimum duration, e.g.,

Jobs	Tasks
a	$\langle 1, 2 \rangle, \langle 2, 6 \rangle$
b	$\langle 2, 5 \rangle, \langle 1, 3 \rangle, \langle 2, 3 \rangle$
c	$\langle 2, 4 \rangle$
d	$\langle 1, 5 \rangle, \langle 2, 2 \rangle$

Planning

Given c cities, t trucks each located at a specific city, and p packages each with a source city and a destination city.

In each step, packages can be loaded and unloaded, or the trucks can be driven from one city to another.

Find a plan with a minimum number of steps for delivering the packages from source to destination.

For each step i, we have Booleans: *location*(t, c, i), *at*(p, c, i), and *on*(p, t, i).

Domain constraints assert that a package can be either on one truck or at a city, a package can be loaded or unloaded from a truck to a city only if the truck is at the city, etc.

Soft Constraints

Given a circuit with a battery B in series with two lamps L_1 and L_2 in parallel. If the battery is normal, then each lamp will light up if it is normal. Lamp L_1 and L_2 do not light up. Minimize abnormality.

The constraints can be expressed as

$\neg ab(L_1)$,

$\neg ab(L_2)$,

$\neg ab(B)$,

$ab(B) \vee ab(L_1) \vee on(L_1)$,

$\neg ab(B)$,

$ab(B) \vee ab(L_2) \vee on(L_2)$,

$\neg on(L_1)$,

$\neg on(L_2)$

AllSAT

The set of *all* satisfying assignments is useful for some applications.

Add a field B to collect the blocking clauses corresponding to the assignments.

For input $\neg a \vee b, c$, the first assignment yields $M = c; a, b$. Add the negation $\neg c \vee \neg a \vee \neg b$ as a blocking clause to B and continue. (This could be reduced to $\neg c \vee \neg b$.)

The next assignment $M' = c; a, \neg b$ generates a confict, so we add the conflict clause $\neg c \vee \neg a$ to C.

Next, $c, \neg a; b$ is a satisfying assignment, so $\neg c \vee a \vee \neg b$ is added to B. Finally, $c, \neg a, \neg b$ is also satisfying, and hence $\neg c \vee a \vee b$ is added to B.

There is a conflict at level 0, and $\neg \bigwedge B$ is the required DNF.

MaxSAT

With soft constraints, all constraints may not be satisfiable, but the goal is to satisfy as many constraints as possible.

Each constraint A_i can be augmented as $a_i \vee A_i$, for a fresh variable a_i.

We can add constraints indicating that at most k of the a_i literals can be assigned $\top$.

By shrinking k, we can determine the minimal value of k.

Weighted MaxSAT can be solved similarly.

More generally, pseudo-Boolean constraints $\Sigma_i w_i * a_i \leq k$ can be encoded.

SMT Applications

- **Test generation:** Find assignments to the individual variables satisfying a path constraint in a program.

- **Infinite-state bounded model checking:** BMC for programs with assignments, unbounded arithmetic, arrays, datatypes, and timers.

- **Predicate abstraction and abstract reachability:** For an atom substitution γ and formula ϕ, find Boolean formula $\hat{\phi}$ such that $\phi \Rightarrow \gamma(\hat{\phi})$.

- Scheduling, planning, constraint solving, and MaxSAT in unbounded domains.

Yices

Yices is a high-performance SMT solver that supports

1. An *expressive language* with higher-order types, dependent types, and predicate subtypes.
2. A *combination of theories* including uninterpreted functions, linear arithmetic, records, tuples, datatypes, arrays, and bit-vectors.
3. A *command language* with incremental definitions, assertions, context creation and examination, pushing/popping contexts, and MaxSAT.

Yices is integrated with SAL and PVS, and is used in hardware/software verification, bounded model checking, planning, probabilistic consistency using MaxSAT, concolic execution.

Yices Examples: Peterson

```
(define c1::(-> int bool))
(define r1::(-> int bool))
(define t::(-> int bool))
(define c2::(-> int bool))
(define r2::(-> int bool))

(define I::bool (and (not (c1 0)) (not (c2 0))(not (r1 0))
                     (not (r2 0))(not (t  0))))

(define N1::(-> int bool) ...)

(define N2::(-> int bool) ...)

(define N::(-> int bool) ...)
```

Yices Example: Property

```
(define safe::(-> int bool) (lambda (i::int)(not (and (c1 i)(c2 i)))))

(define iter_N::(-> int int bool) (lambda (i::int j::int)
   (if (<= i 0) (N j) (and (N (+ i j)) (iter_N (- i 1) j)))))

(define safeto::(-> int int bool)(lambda (i::int j::int)
   (if (<= i 0) (safe j)(and (safe (+ i j))(safeto (- i 1) j)))))

(define PBMC::(-> int bool) (lambda (i::int)
   (and I (iter_N i 0) (not (safeto (+ i 1) 0)))))

(define PIND::(-> int bool)(lambda (i::int)(exists (j::int)
   (and (iter_N i j) (safeto i j)(not (safeto (+ i 1) j))))))

(assert (or (PBMC 3) (PIND 3)))
(check)
```

Yices Example: Lightbulb

```
(define-type bulb (scalar lb1 lb2))
(define abl::(-> bulb bool))
(define abb::bool)
(define on::(-> bulb bool))
(assert+ (not abb) 1)
(assert+ (not (abl lb1)) 1)
(assert+ (not (abl lb2)) 1)
(assert (or abb (abl lb1) (on lb1)))
(assert (or abb (abl lb2) (on lb2)))
(assert (not (on lb2)))
(assert (not (on lb1)))
(max-sat)
```

Lightbulb output

```
sat
unsatisfied assertion ids: 1
(= abb true)
(= (on lb1) false)
(= (on lb2) false)
(= (abl lb1) false)
(= (abl lb2) false)
(= (on lb1) false)
(= (on lb2) false)

cost: 1
```

Yices Example: Arrays

```
(not
 (forall (?i Int) (?pp Queue)(?aa Array)(?perm Array)(?ee Array)
         (?newperm Array)
         (implies (and (= ?ee (store (store (elems ?pp)
                                            (- ?i 1)
                                            (select (elems ?pp) ?i))
                                     ?i (select (elems ?pp) (- ?i 1))))
                       (= ?newperm (store (store ?perm
                                                 (- ?i 1)
                                                 (select ?perm ?i))
                                          ?i (select ?perm (- ?i 1))))
                       (forall (?i Int) (= (select ?aa (select ?perm ?i))
                                           (select (elems ?pp) ?i))))
                  (forall (?i Int) (= (select ?aa (select ?newperm ?i))
                                      (select ?ee ?i))))))
```

Conclusions

Powerful, mature, and versatile tools like SMT solvers can now be exploited in very useful ways.

Applications include verification, test generation, model checking, theorem proving, abstraction, scheduling, planning, and soft constraint solving.

These tools can be used either as blackboxes, libraries, scripts, or interactively.

The construction and application of satisfiability procedures is an active research area with exciting challenges (nonlinear arithmetic, quantifier reasoning, scalability, interfaces, integration).