SMT Solvers

Theory & Practice

Leonardo de Moura

leonardo@microsoft.com

Microsoft Research

Credits

- Slides inspired by previous presentations by:

  Clark Barrett, Harald Ruess, Natarajan Shankar, Cesare Tinelli, Ashish Tiwari

- Special thanks to:

  Clark Barrett, Cesare Tinelli (for contributing some of the material) and the FMCAD PC (for the invitation).

Introduction

- Industry tools rely on powerful verification engines.
  - Boolean satisfiability (SAT) solvers.
  - Binary decision diagrams (BDDs).
- *Satisfiability Modulo Theories (SMT)*
  - The next generation of verification engines.
  - *SAT solvers + Theories*
    - Arithmetic
    - Arrays
    - Uninterpreted Functions
  - Some problems are more naturally expressed in SMT.
  - More automation.

Applications

- Extended Static Checking.
  - *Microsoft Spec# and ESP.*
  - ESC/Java
- Predicate Abstraction.
  - *Microsoft SLAM/SDV (device driver verification).*
- Bounded Model Checking (BMC) & k-induction.
- Test-case generation.
  - *Microsoft MUTT.*
- Symbolic Simulation.
- Planning & Scheduling.
- Equivalence checking.

SMT-Solvers & SMT-Lib & SMT-Comp

- SMT-Solves:

  Ario, Barcelogic, CVC, CVC Lite, CVC3, ExtSAT, Harvey, HTP, *ICS (SRI)*, Jat, MathSAT, Sateen, Simplify, STeP, STP, SVC, TSAT, UCLID, *Yices (SRI)*, Zap (Microsoft), *Z3 (Microsoft)*

- SMT-Lib: library of benchmarks
  `http://goedel.cs.uiowa.edu/smtlib/`
- SMT-Comp: annual SMT-Solver competition.

Roadmap

- Background
- Theories
- Combination of Theories
- SAT + Theories
- Decision Procedures for Specific Theories
- Applications

Language: Signatures

- A *signature* Σ is a finite set of:
  - Function symbols: $\Sigma_F = \{f, g, \ldots\}$.
  - Predicate symbols: $\Sigma_P = \{P, Q, \ldots\}$.
  - and an *arity* function: $\Sigma \mapsto N$
- Function symbols with arity 0 are called *constants*.
- A countable set $\mathcal{V}$ of *variables* disjoint of Σ.

Language: Terms

- The set $T(\Sigma, \mathcal{V})$ of *terms* is the smallest set such that:
  - $\mathcal{V} \subset T(\Sigma, \mathcal{V})$
  - $f(t_1, \ldots, t_n) \in T(\Sigma, \mathcal{V})$ whenever $f \in \Sigma_F, t_1, \ldots, t_n \in T(\Sigma, \mathcal{V})$ and *arity*$(f) = n$.
- The set of *ground terms* is defined as $T(\Sigma, \emptyset)$.

Language: Atomic Formulas

- $P(t_1, \ldots, t_n)$ is an *atomic formula* whenever $P \in \Sigma_P$, *arity*$(P) = n$, and $t_1, \ldots, t_n \in T(\Sigma, \mathcal{V})$.
- *true* and *false* are atomic formulas.
- If $t_1, \ldots, t_n$ are ground terms, then $P(t_1, \ldots, t_n)$ is called a *ground (atomic) formula*.
- We assume that the binary predicate $=$ is present in Σ_P.
- A *literal* is an atomic formula or its negation.

Language: Quantifier Free Formulas

- The set *QFF*$(\Sigma, \mathcal{V})$ of *quantifier free formulas* is the smallest set such that:
  - Every atomic formulas is in *QFF*$(\Sigma, \mathcal{V})$.
  - If $\phi \in$ *QFF*$(\Sigma, \mathcal{V})$, then $\neg\phi \in$ *QFF*$(\Sigma, \mathcal{V})$.
  - If $\phi_1, \phi_2 \in$ *QFF*$(\Sigma, \mathcal{V})$, then

$$\begin{array}{rcl} \phi_1 \wedge \phi_2 & \in & \mathit{QFF}(\Sigma, \mathcal{V}) \\ \phi_1 \vee \phi_2 & \in & \mathit{QFF}(\Sigma, \mathcal{V}) \\ \phi_1 \Rightarrow \phi_2 & \in & \mathit{QFF}(\Sigma, \mathcal{V}) \\ \phi_1 \Leftrightarrow \phi_2 & \in & \mathit{QFF}(\Sigma, \mathcal{V}) \end{array}$$

Language: Formulas

- The set of *first-order formulas* is the closure of *QFF*$(\Sigma, \mathcal{V})$ under existential ($\exists$) and universal ($\forall$) quantification.
- *Free* (occurrences) of *variables* in a formula are those not bound by a quantifier.
- A *sentence* is a first-order formula with no free variables.

Theories

- A *(first-order) theory* $\mathcal{T}$ (over a signature Σ) is a set of (deductively closed) sentences (over Σ and $\mathcal{V}$).
- Let *DC*(Γ) be the deductive closure of a set of sentences Γ.
  - For every theory $\mathcal{T}$, *DC*$(\mathcal{T}) = \mathcal{T}$.
- A theory $\mathcal{T}$ is *consistent* if *false* $\notin \mathcal{T}$.
- We can view a (first-order) theory $\mathcal{T}$ as the class of all *models* of $\mathcal{T}$ (due to completeness of first-order logic).

Models (Semantics)

- A model M is defined as:
  - Domain S: set of elements.
  - Interpretation $f^M : S^n \mapsto S$ for each $f \in \Sigma_F$ with *arity*$(f) = n$.
  - Interpretation $P^M \subseteq S^n$ for each $P \in \Sigma_P$ with *arity*$(P) = n$.
  - Assignment $x^M \in S$ for every variable $x \in \mathcal{V}$.
- A formula ϕ is true in a model M if it evaluates to true under the given interpretations over the domain S.
- M is a *model for the theory* $\mathcal{T}$ if all sentences of $\mathcal{T}$ are true in M.

Satisfiability and Validity

- A formula $\phi(\vec{x})$ is *satisfiable* in a theory $\mathcal{T}$ if there is a model of *DC*$(\mathcal{T} \cup \exists\vec{x}.\phi(\vec{x}))$. That is, there is a model M for $\mathcal{T}$ in which $\phi(\vec{x})$ evaluates to true, denoted by,

$$M \models_{\mathcal{T}} \phi(\vec{x})$$

- This is also called *$\mathcal{T}$-satisfiability*.
- A formula $\phi(\vec{x})$ is *valid* in a theory $\mathcal{T}$ if $\forall\vec{x}.\phi(\vec{x}) \in \mathcal{T}$. That is $\phi(\vec{x})$ evaluates to true in every model M of $\mathcal{T}$.
- *$\mathcal{T}$-validity* is denoted by $\models_{\mathcal{T}} \phi(\vec{x})$.
- The *quantifier free $\mathcal{T}$-satisfiability problem* restricts ϕ to be *quantifier free*.

Checking validity

- Checking the *validity* of ϕ in a theory $\mathcal{T}$ is:

  $\equiv \mathcal{T}$-satisfiability of $\neg\phi$

  $\equiv \mathcal{T}$-satisfiability of $\vec{Q}\vec{x}.\phi_1$ (PNF of $\neg\phi$)

  $\equiv \mathcal{T}$-satisfiability of $\forall\vec{x}.\phi_1$ (Skolemize)

  $\equiv \mathcal{T}$-satisfiability of ϕ_2 (Instantiate)

  $\equiv \mathcal{T}$-satisfiability of $\bigvee_i \psi_i$ (DNF of ϕ_2)

  $\equiv \mathcal{T}$-satisfiability of every ψ_i

- ψ_i is a conjunction of literals.

Roadmap

- Background
- Theories
- Combination of Theories
- SAT + Theories
- Decision Procedures for Specific Theories
- Applications

Pure Theory of Equality (EUF)

- The theory $\mathcal{T}_\mathcal{E}$ of equality is the theory $DC(\emptyset)$.
- The exact set of sentences of $\mathcal{T}_\mathcal{E}$ depends on the *signature* in question.
- The theory does not restrict the possibles values of the symbols in its signature in any way. For this reason, it is sometimes called the theory of *equality and uninterpreted functions*.
- The satisfiability problem for $\mathcal{T}_\mathcal{E}$ is the satisfiability problem for first-order logic, which is undecidable.
- The satisfiability problem for conjunction of literals in $\mathcal{T}_\mathcal{E}$ is decidable in polynomial time using *congruence closure*.

Linear Integer Arithmetic

- $\Sigma_P = \{\leq\}, \Sigma_F = \{0, 1, +, -\}$.
- Let $M_{\mathcal{LIA}}$ be the standard model of integers.
- Then $\mathcal{T}_{\mathcal{LIA}}$ is defined to be the set of all Σ sentences true in the model $M_{\mathcal{LIA}}$.
- As showed by Presburger, the general satisfiability problem for $\mathcal{T}_{\mathcal{LIA}}$ is decidable, but its complexity is triply-exponential.
- The quantifier free satisfiability problem is NP-complete.
- Remark: non-linear integer arithmetic is undecidable even for the quantifier free case.

Linear Real Arithmetic

- The general satisfiability problem for $\mathcal{T}_{\mathcal{LRA}}$ is decidable, but its complexity is doubly-exponential.
- The quantifier free satisfiability problem is solvable in polynomial time, though exponential methods (Simplex) tend to perform best in practice.

Difference Logic

- *Difference logic* is a fragment of linear arithmetic.
- Atoms have the form: $x - y \leq c$.
- Most linear arithmetic atoms found in hardware and software verification are in this fragment.
- The quantifier free satisfiability problem is solvable in $O(nm)$.

Theory of Arrays

- $\Sigma_P = \emptyset$, $\Sigma_F = \{$*read*, *write*$\}$.
- Non-extensional arrays
  - Let $\Lambda_{\mathcal{A}}$ be the following axioms:

$$\forall a, i, v.\; read(write(a, i, v), i) = v$$

$$\forall a, i, j, v.\; i \neq j \Rightarrow read(write(a, i, v), j) = read(a, j)$$

  - $\mathcal{T}_{\mathcal{A}} = DC(\Lambda_{\mathcal{A}})$
- For extensional arrays, we need the following extra axiom:

$$\forall a, b.\; (\forall i. read(a, i) = read(b, i)) \Rightarrow a = b$$

- The satisfiability problem for $\mathcal{T}_{\mathcal{A}}$ is undecidable, the quantifier free case is NP-complete.

Other theories

- Bit-vectors
- Partial orders
- Tuples & Records
- Algebraic data types
- . . .

Roadmap

- Background
- Theories
- Combination of Theories
- SAT + Theories
- Decision Procedures for Specific Theories
- Applications

Combination of Theories

- In practice, we need a combination of theories.
- Examples:
  - $x + 2 = y \Rightarrow f(\mathit{read}(\mathit{write}(a, x, 3), y - 2)) = f(y - x + 1)$
  - $f(f(x) - f(y)) \neq f(z), x + z \leq y \leq x \Rightarrow z < 0$
- Given

$$\begin{array}{rcl} \Sigma & = & \Sigma_1 \cup \Sigma_2 \\ \mathcal{T}_1, \mathcal{T}_2 & : & \textit{theories over } \Sigma_1, \Sigma_2 \\ \mathcal{T} & = & \mathit{DC}(\mathcal{T}_1 \cup \mathcal{T}_2) \end{array}$$

- Is $\mathcal{T}$ consistent?
- Given satisfiability procedures for conjunction of literals of $\mathcal{T}_1$ and $\mathcal{T}_2$, how to decide the satisfiability of $\mathcal{T}$?

Preamble

- Disjoint signatures: $\Sigma_1 \cap \Sigma_2 = \emptyset$.
- Stably-Infinite Theories.
- Convex Theories.

Stably-Infinite Theories

- A theory is *stably infinite* if every satisfiable QFF is satisfiable in an infinite model.
- Example. Theories with only finite models are not stably infinite. $\mathcal{T}_2 = DC(\forall x, y, z.\, (x = y) \vee (x = z) \vee (y = z))$.
- Is this a problem in practice? (We want to support the “finite types” found in our programming languages)

Stably-Infinite Theories

- A theory is *stably infinite* if every satisfiable QFF is satisfiable in an infinite model.
- Example. Theories with only finite models are not stably infinite.
  $\mathcal{T}_2 = DC(\forall x, y, z.\ (x = y) \lor (x = z) \lor (y = z))$.
- Is this a problem in practice? (We want to support the "finite types" found in our programming languages)
- Answer: *No*. $\mathcal{T}_2$ is not useful in practice. Add a predicate $in_2(x)$ (intuition: x is an element of the "finite type").

$$\mathcal{T}_2{}' = DC(\forall x, y, z.\ in_2(x) \land in_2(y) \land in_2(z) \Rightarrow \\ (x = y) \lor (x = z) \lor (y = z))$$

- $\mathcal{T}_2{}'$ *is stably infinite*.

Stably-Infinite Theories (cont.)

- *The union of two consistent, disjoint, stably infinite theories is consistent.*

Convexity

- A theory $\mathcal{T}$ is *convex* iff

  for all finite sets Γ of literals and

  for all non-empty disjunctions $\bigvee_{i \in I} x_i = y_i$ of variables,

  $\Gamma \models_{\mathcal{T}} \bigvee_{i \in I} x_i = y_i$ iff $\Gamma \models_{\mathcal{T}} x_i = y_i$ for some $i \in I$.

- Every convex theory $\mathcal{T}$ with non trivial models (i.e., $\models_T \exists x, y.\ x \neq y$) is stably infinite.

- All *Horn* theories are convex – this includes all (conditional) equational theories.

- *Linear rational arithmetic is convex.*

Convexity (cont.)

- *Many theories are not convex:*
  - Linear integer arithmetic.

$$1 \leq x \leq 3 \models x = 1 \vee x = 2 \vee x = 3$$

  - Nonlinear arithmetic.

$$x^2 = 1, y = 1, z = -1 \models x = y \vee x = z$$

  - Theory of Bit-vectors.
  - Theory of Arrays.

$$v_1 = \mathit{read}(\mathit{write}(a, i, v_2), j), v_3 = \mathit{read}(a, j) \models$$

$$v_1 = v_2 \vee v_1 = v_3$$

Convexity: Example

- Let $\mathcal{T} = \mathcal{T}_1 \cup \mathcal{T}_2$, where $\mathcal{T}_1$ is EUF ($O(nlog(n))$) and $\mathcal{T}_2$ is IDL ($O(nm)$).
- *$\mathcal{T}_2$ is not convex*.
- *Satisfiability is NP-Complete for $\mathcal{T} = \mathcal{T}_1 \cup \mathcal{T}_2$.*
  - Reduce 3CNF satisfiability to $\mathcal{T}$-satisfiability.
  - For each boolean variable p_i add the atomic formulas: $0 \leq x_i$, $x_i \leq 1$.
  - For a clause $p_1 \vee \neg p_2 \vee p_3$ add the atomic formula: $f(x_1, x_2, x_3) \neq f(0, 1, 0)$

Nelson-Oppen Combination

- Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be consistent, stably infinite theories over disjoint (countable) signatures. Assume satisfiability of conjunction of literals can decided in $O(T_1(n))$ and $O(T_2(n))$ time respectively. Then,
  1. The combined theory $\mathcal{T}$ is consistent and stably infinite.
  2. Satisfiability of quantifier free conjunction of literals in $\mathcal{T}$ can be decided in $O(2^{n^2} \times (T_1(n) + T_2(n))$.
  3. If $\mathcal{T}_1$ and $\mathcal{T}_2$ are convex, then so is $\mathcal{T}$ and satisfiability in $\mathcal{T}$ is in $O(n^4 \times (T_1(n) + T_2(n)))$.

Nelson-Oppen Combination Procedure

- The combination procedure:

  Initial State: ϕ is a conjunction of literals over $\Sigma_1 \cup \Sigma_2$.

  Purification: Preserving satisfiability transform ϕ into $\phi_1 \wedge \phi_2$, such that, $\phi_i \in \Sigma_i$.

  Interaction: Guess a partition of $\mathcal{V}(\phi_1) \cap \mathcal{V}(\phi_2)$ into disjoint subsets. Express it as conjunction of literals ψ.
  Example. The partition $\{x_1\}, \{x_2, x_3\}, \{x_4\}$ is represented as $x_1 \neq x_2, x_1 \neq x_4, x_2 \neq x_4, x_2 = x_3$.

  Component Procedures : Use individual procedures to decide whether $\phi_i \wedge \psi$ is satisfiable.

  Return: If both return yes, return yes. No, otherwise.

Purification

- Purification:

  $\phi \wedge P(\ldots, s[t], \ldots) \rightsquigarrow \phi \wedge P(\ldots, s[x], \ldots) \wedge x = t$,

  t is not a variable.

- *Purification is satisfiability preserving and terminating.*

Purification

- Purification:

  $\phi \wedge P(\ldots, s[t], \ldots) \rightsquigarrow \phi \wedge P(\ldots, s[x], \ldots) \wedge x = t$,

  t is not a variable.

- *Purification is satisfiability preserving and terminating.*

- Example:

$$f(x - 1) - 1 = x, f(y) + 1 = y \rightsquigarrow$$

Purification

- Purification:

  $\phi \wedge P(\ldots, s[t], \ldots) \rightsquigarrow \phi \wedge P(\ldots, s[x], \ldots) \wedge x = t$,

  t is not a variable.

- *Purification is satisfiability preserving and terminating.*

- Example:

$$f(x-1) - 1 = x, f(y) + 1 = y \rightsquigarrow$$

$$f(u_1) - 1 = x, f(y) + 1 = y, u_1 = x - 1 \rightsquigarrow$$

Purification

- Purification:

  $\phi \wedge P(\ldots, s[t], \ldots) \rightsquigarrow \phi \wedge P(\ldots, s[x], \ldots) \wedge x = t$,

  t is not a variable.

- *Purification is satisfiability preserving and terminating.*

- Example:

$$f(x - 1) - 1 = x, f(y) + 1 = y \rightsquigarrow$$
$$f(u_1) - 1 = x, f(y) + 1 = y, u_1 = x - 1 \rightsquigarrow$$
$$u_2 - 1 = x, f(y) + 1 = y, u_1 = x - 1, u_2 = f(u_1) \rightsquigarrow$$

Purification

- Purification:

  $\phi \wedge P(\ldots, s[t], \ldots) \rightsquigarrow \phi \wedge P(\ldots, s[x], \ldots) \wedge x = t$,

  t is not a variable.

- *Purification is satisfiability preserving and terminating.*

- Example:

$$f(x-1) - 1 = x, f(y) + 1 = y \rightsquigarrow$$

$$f(u_1) - 1 = x, f(y) + 1 = y, u_1 = x - 1 \rightsquigarrow$$

$$u_2 - 1 = x, f(y) + 1 = y, u_1 = x - 1, u_2 = f(u_1) \rightsquigarrow$$

$$u_2 - 1 = x, u_3 + 1 = y, u_1 = x - 1, u_2 = f(u_1), u_3 = f(y)$$

Purification

- Purification:

  $\phi \wedge P(\ldots, s[t], \ldots) \rightsquigarrow \phi \wedge P(\ldots, s[x], \ldots) \wedge x = t$,

  t is not a variable.

- *Purification is satisfiability preserving and terminating.*

- Example:

$$f(x-1) - 1 = x, f(y) + 1 = y \rightsquigarrow$$

$$f(u_1) - 1 = x, f(y) + 1 = y, u_1 = x - 1 \rightsquigarrow$$

$$u_2 - 1 = x, f(y) + 1 = y, u_1 = x - 1, u_2 = f(u_1) \rightsquigarrow$$

$$u_2 - 1 = x, u_3 + 1 = y, u_1 = x - 1, u_2 = f(u_1), u_3 = f(y)$$

Purification (cont.)

- As most of the SMT developers will tell you, the purification step is not really necessary.
- Given a set of mixed (impure) literal Γ, define a *shared term* to be any term in Γ which is *alien* in some literal or sub-term in Γ.
- In our examples, these were the terms replaced by constants.
- Assume that each satisfiability procedure treats alien terms as constants.

NO procedure: soundness

- Each step is satisfiability preserving.
- Say ϕ is satisfiable (in the combination).
  - Purification: $\phi_1 \wedge \phi_2$ is satisfiable.

NO procedure: soundness

- Each step is satisfiability preserving.
- Say ϕ is satisfiable (in the combination).
  - Purification: $\phi_1 \wedge \phi_2$ is satisfiable.
  - Iteration: for some partition ψ, $\phi_1 \wedge \phi_2 \wedge \psi$ is satisfiable.

NO procedure: soundness

- Each step is satisfiability preserving.
- Say ϕ is satisfiable (in the combination).
  - Purification: $\phi_1 \wedge \phi_2$ is satisfiable.
  - Iteration: for some partition ψ, $\phi_1 \wedge \phi_2 \wedge \psi$ is satisfiable.
  - Component procedures: $\phi_1 \wedge \psi$ and $\phi_2 \wedge \psi$ are both satisfiable in component theories.

NO procedure: soundness

- Each step is satisfiability preserving.
- Say ϕ is satisfiable (in the combination).
  - Purification: $\phi_1 \wedge \phi_2$ is satisfiable.
  - Iteration: for some partition ψ, $\phi_1 \wedge \phi_2 \wedge \psi$ is satisfiable.
  - Component procedures: $\phi_1 \wedge \psi$ and $\phi_2 \wedge \psi$ are both satisfiable in component theories.
  - Therefore, if the procedure return unsatisfiable, then ϕ is unsatisfiable.

NO procedure: correctness

- Suppose the procedure returns satisfiable.
  - Let ψ be the partition and A and B be models of $\mathcal{T}_1 \wedge \phi_1 \wedge \psi$ and $\mathcal{T}_2 \wedge \phi_2 \wedge \psi$.

NO procedure: correctness

- Suppose the procedure returns satisfiable.
  - Let ψ be the partition and A and B be models of $\mathcal{T}_1 \wedge \phi_1 \wedge \psi$ and $\mathcal{T}_2 \wedge \phi_2 \wedge \psi$.
  - The component theories are stably infinite. So, assume the models are infinite (of same cardinality).

NO procedure: correctness

- Suppose the procedure returns satisfiable.
  - Let ψ be the partition and A and B be models of $\mathcal{T}_1 \wedge \phi_1 \wedge \psi$ and $\mathcal{T}_2 \wedge \phi_2 \wedge \psi$.
  - The component theories are stably infinite. So, assume the models are infinite (of same cardinality).
  - Let h be a bijection between S_A and S_B such that $h(x^A) = x^B$ for each shared variable.

NO procedure: correctness

- Suppose the procedure returns satisfiable.
  - Let ψ be the partition and A and B be models of $\mathcal{T}_1 \wedge \phi_1 \wedge \psi$ and $\mathcal{T}_2 \wedge \phi_2 \wedge \psi$.
  - The component theories are stably infinite. So, assume the models are infinite (of same cardinality).
  - Let h be a bijection between S_A and S_B such that $h(x^A) = x^B$ for each shared variable.
  - Extend B to $\bar{B}$ by interpretations of symbols in Σ_1: $f^{\bar{B}}(b_1, \ldots, b_n) = h(f^A(h^{-1}(b_1), \ldots, h^{-1}(b_n)))$

NO procedure: correctness

- Suppose the procedure returns satisfiable.
  - Let ψ be the partition and A and B be models of $\mathcal{T}_1 \wedge \phi_1 \wedge \psi$ and $\mathcal{T}_2 \wedge \phi_2 \wedge \psi$.
  - The component theories are stably infinite. So, assume the models are infinite (of same cardinality).
  - Let h be a bijection between S_A and S_B such that $h(x^A) = x^B$ for each shared variable.
  - Extend B to $\bar{B}$ by interpretations of symbols in Σ_1: $f^{\bar{B}}(b_1, \ldots, b_n) = h(f^A(h^{-1}(b_1), \ldots, h^{-1}(b_n)))$
  - $\bar{B}$ *is a model of:* $\mathcal{T}_1 \wedge \phi_1 \wedge \mathcal{T}_2 \wedge \phi_2 \wedge \psi$

NO deterministic procedure

- Instead of *guessing*, we can *deduce* the equalities to be shared.

  Purification: no changes.

  Interaction: Deduce an equality $x = y$:

$$\mathcal{T}_1 \vdash (\phi_1 \Rightarrow x = y)$$

  Update $\phi_2 := \phi_2 \wedge x = y$. And vice-versa. Repeat until no further changes.

  Component Procedures : Use individual procedures to decide whether ϕ_i is satisfiable.

- Remark: $\mathcal{T}_i \vdash (\phi_i \Rightarrow x = y)$ iff $\phi_i \wedge x \neq y$ is not satisfiable in $\mathcal{T}_i$.

NO deterministic procedure: correctness

- Assume the theories are convex.
  - Suppose ϕ_i is satisfiable.

NO deterministic procedure: correctness

- Assume the theories are convex.
  - Suppose ϕ_i is satisfiable.
  - Let E be the set of equalities $x_j = x_k$ ($j \neq k$) such that, $\mathcal{T}_i \not\models \phi_i \Rightarrow x_j = x_k$.

NO deterministic procedure: correctness

- Assume the theories are convex.
  - Suppose ϕ_i is satisfiable.
  - Let E be the set of equalities $x_j = x_k$ ($j \neq k$) such that, $\mathcal{T}_i \not\models \phi_i \Rightarrow x_j = x_k$.
  - By convexity, $\mathcal{T}_i \not\models \phi_i \Rightarrow \bigvee_E x_j = x_k$.

NO deterministic procedure: correctness

- Assume the theories are convex.
  - Suppose ϕ_i is satisfiable.
  - Let E be the set of equalities $x_j = x_k$ ($j \neq k$) such that, $\mathcal{T}_i \not\models \phi_i \Rightarrow x_j = x_k$.
  - By convexity, $\mathcal{T}_i \not\models \phi_i \Rightarrow \bigvee_E x_j = x_k$.
  - $\phi_i \wedge \bigwedge_E x_j \neq x_k$ is satisfiable.

NO deterministic procedure: correctness

- Assume the theories are convex.
  - Suppose ϕ_i is satisfiable.
  - Let E be the set of equalities $x_j = x_k$ ($j \neq k$) such that, $\mathcal{T}_i \not\models \phi_i \Rightarrow x_j = x_k$.
  - By convexity, $\mathcal{T}_i \not\models \phi_i \Rightarrow \bigvee_E x_j = x_k$.
  - $\phi_i \wedge \bigwedge_E x_j \neq x_k$ is satisfiable.
  - The proof now is identical to the nondeterministic case.

NO procedure: example

$$x + 2 = y \wedge f(\textit{read}(\textit{write}(a, x, 3), y - 2)) \neq f(y - x + 1)$$

$\mathcal{T}_{\mathcal{E}}$	$\mathcal{T}_{\mathcal{LA}}$	$\mathcal{T}_{\mathcal{A}}$

Purifying

NO procedure: example

$$f(\textit{read}(\textit{write}(a, x, \mathit{3}), y - 2)) \neq f(y - x + 1)$$

$\mathcal{T}_{\mathcal{E}}$	$\mathcal{T}_{\mathcal{LA}}$	$\mathcal{T}_{\mathcal{A}}$
	$x + 2 = y$	

Purifying

NO procedure: example

$$f(\textit{read}(\textit{write}(a, x, u_1), y - 2)) \neq f(y - x + 1)$$

$\mathcal{T}_{\mathcal{E}}$	$\mathcal{T}_{\mathcal{LA}}$	$\mathcal{T}_{\mathcal{A}}$
	$x + 2 = y$ $u_1 = 3$	

Purifying

NO procedure: example

$$f(\mathit{read}(\mathit{write}(a, x, u_1), u_2)) \neq f(y - x + 1)$$

$\mathcal{T}_{\mathcal{E}}$	$\mathcal{T}_{\mathcal{LA}}$	$\mathcal{T}_{\mathcal{A}}$
	$x + 2 = y$ $u_1 = 3$ $u_2 = y - 2$	

Purifying

NO procedure: example

$f(u_3) \neq f(y - x + 1)$

$\mathcal{T}_{\mathcal{E}}$	$\mathcal{T}_{\mathcal{LA}}$	$\mathcal{T}_{\mathcal{A}}$
	$x + 2 = y$ $u_1 = 3$ $u_2 = y - 2$	$u_3 =$ $\mathit{read}(\mathit{write}(a, x, u_1), u_2)$

Purifying

NO procedure: example

$f(u_3) \neq f(u_4)$

$\mathcal{T}_{\mathcal{E}}$	$\mathcal{T}_{\mathcal{LA}}$	$\mathcal{T}_{\mathcal{A}}$
	$x + 2 = y$ $u_1 = 3$ $u_2 = y - 2$ $u_4 = y - x + 1$	$u_3 =$ $\textit{read}(\textit{write}(a, x, u_1), u_2)$

Purifying

NO procedure: example

$\mathcal{T}_{\mathcal{E}}$	$\mathcal{T}_{\mathcal{LA}}$	$\mathcal{T}_{\mathcal{A}}$
$f(u_3) \neq f(u_4)$	$x + 2 = y$ $u_1 = 3$ $u_2 = y - 2$ $u_4 = y - x + 1$	$u_3 =$ $\mathit{read}(\mathit{write}(a, x, u_1), u_2)$

Solving $\mathcal{T}_{\mathcal{LA}}$

NO procedure: example

$\mathcal{T}_{\mathcal{E}}$	$\mathcal{T}_{\mathcal{LA}}$	$\mathcal{T}_{\mathcal{A}}$
$f(u_3) \neq f(u_4)$	$y = x + 2$ $u_1 = 3$ $u_2 = x$ $u_4 = 3$	$u_3 =$ $\textit{read}(\textit{write}(a, x, u_1), u_2)$

Propagating $u_2 = x$

NO procedure: example

$\mathcal{T}_{\mathcal{E}}$	$\mathcal{T}_{\mathcal{LA}}$	$\mathcal{T}_{\mathcal{A}}$
$f(u_3) \neq f(u_4)$	$y = x + 2$	$u_3 =$
$u_2 = x$	$u_1 = 3$	*read*$($*write*$(a, x, u_1), u_2)$
	$u_2 = x$	$u_2 = x$
	$u_4 = 3$	

Solving $\mathcal{T}_{\mathcal{A}}$

NO procedure: example

$\mathcal{T}_{\mathcal{E}}$	$\mathcal{T}_{\mathcal{LA}}$	$\mathcal{T}_{\mathcal{A}}$
$f(u_3) \neq f(u_4)$	$y = x + 2$	$u_3 = u_1$
$u_2 = x$	$u_1 = 3$	$u_2 = x$
	$u_2 = x$	
	$u_4 = 3$	

Propagating $u_3 = u_1$

NO procedure: example

$\mathcal{T}_{\mathcal{E}}$	$\mathcal{T}_{\mathcal{LA}}$	$\mathcal{T}_{\mathcal{A}}$
$f(u_3) \neq f(u_4)$	$y = x + 2$	$u_3 = u_1$
$u_2 = x$	$u_1 = 3$	$u_2 = x$
$u_3 = u_1$	$u_2 = x$	
	$u_4 = 3$	
	$u_3 = u_1$	

Propagating $u_1 = u_4$

NO procedure: example

$\mathcal{T}_{\mathcal{E}}$	$\mathcal{T}_{\mathcal{LA}}$	$\mathcal{T}_{\mathcal{A}}$
$f(u_3) \neq f(u_4)$	$y = x + 2$	$u_3 = u_1$
$u_2 = x$	$u_1 = 3$	$u_2 = x$
$u_3 = u_1$	$u_2 = x$	
$u_4 = u_1$	$u_4 = 3$	
	$u_3 = u_1$	

Congruence $u_3 = u_1 \wedge u_4 = u_1 \Rightarrow f(u_3) = f(u_4)$

NO procedure: example

$\mathcal{T}_{\mathcal{E}}$	$\mathcal{T}_{\mathcal{LA}}$	$\mathcal{T}_{\mathcal{A}}$
$f(u_3) \neq f(u_4)$	$y = x + 2$	$u_3 = u_1$
$u_2 = x$	$u_1 = 3$	$u_2 = x$
$u_3 = u_1$	$u_2 = x$	
$u_4 = u_1$	$u_4 = 3$	
$f(u_3) = f(u_4)$	$u_3 = u_1$	

Unsatisfiable!

Reduction Functions

- A *reduction function* reduces the satisfiability of a complex theory to the satisfiability problem of a simpler theory.
- *Ackerman reduction* is used to remove uninterpreted functions.
  - For each application $f(\vec{a})$ in ϕ create a fresh variable $f_{\vec{a}}$.
  - For each pair of applications $f(\vec{a})$, $f(\vec{c})$ in ϕ add the formula $\vec{a} = \vec{c} \Rightarrow f_{\vec{a}} = f_{\vec{c}}$.
  - It is used in some SMT solvers to reduce $\mathcal{T}_{\mathcal{LA}} \cup \mathcal{T}_{\mathcal{E}}$ to $\mathcal{T}_{\mathcal{LA}}$.

Reduction Functions

- Theory of commutative functions.
  - Deductive closure of: $\forall x, y. f(x, y) = f(y, x)$
  - Reduction to $\mathcal{T}_{\mathcal{E}}$.
  - For every $f(a, b)$ in ϕ, do $\phi := \phi \wedge f(a, b) = f(b, a)$.
- Theory of “lists”.
  - Deductive closure of:

$$\forall x, y.\ car(cons(x, y)) = x$$
$$\forall x, y.\ cdr(cons(x, y)) = y$$

  - Reduction to $\mathcal{T}_{\mathcal{E}}$
  - For each term $cons(a, b)$ in ϕ, do
    $\phi := \phi \wedge car(cons(a, b)) = a \wedge cdr(cons(a, b)) = b$.

Roadmap

- Background
- Theories
- Combination of Theories
- SAT + Theories
- Decision Procedures for Specific Theories
- Applications

Breakthrough in SAT solving

- Breakthrough in SAT solving influenced the way SMT solvers are implemented.
- Modern SAT solvers are based on the DPLL algorithm.
- Modern implementations add several sophisticated *search techniques*.
  - Backjumping
  - Learning
  - Restarts
  - Watched literals

The Original DPLL Procedure

- Tries to *build* incrementally a *satisfying truth assignment* M for a CNF formula F.
- M is grown by
  - *deducing* the truth value of a literal from M and F, or
  - *guessing* a truth value.
- If a wrong guess leads to an inconsistency, the procedure *backtracks* and tries the opposite one.

Basic DPLL System – Example

$$\emptyset \;\|\; \overline{1} \vee 2,\; \overline{3} \vee 4,\; \overline{5} \vee \overline{6},\; 6 \vee \overline{5} \vee \overline{2}$$

Basic DPLL System – Example

$$\emptyset \parallel \overline{1} \vee 2,\ \ \overline{3} \vee 4,\ \ \overline{5} \vee \overline{6},\ \ 6 \vee \overline{5} \vee \overline{2} \quad \Longrightarrow \quad (\text{Decide})$$

$$1 \parallel \overline{1} \vee 2,\ \ \overline{3} \vee 4,\ \ \overline{5} \vee \overline{6},\ \ 6 \vee \overline{5} \vee \overline{2}$$

Basic DPLL System – Example

$$\emptyset \parallel \overline{1} \vee 2,\ \ \overline{3} \vee 4,\ \ \overline{5} \vee \overline{6},\ \ 6 \vee \overline{5} \vee \overline{2} \quad \Longrightarrow \quad (\text{Decide})$$

$$\color{red}{1} \parallel \overline{1} \vee 2,\ \ \overline{3} \vee 4,\ \ \overline{5} \vee \overline{6},\ \ 6 \vee \overline{5} \vee \overline{2} \quad \Longrightarrow \quad (\text{UnitProp})$$

$$\color{red}{1}\ 2 \parallel \overline{1} \vee 2,\ \ \overline{3} \vee 4,\ \ \overline{5} \vee \overline{6},\ \ 6 \vee \overline{5} \vee \overline{2}$$

Basic DPLL System – Example

$$\begin{array}{rcl}
\emptyset \;\|\; \overline{1} \vee 2,\; \overline{3} \vee 4,\; \overline{5} \vee \overline{6},\; 6 \vee \overline{5} \vee \overline{2} & \Longrightarrow & (\text{Decide}) \\
\color{red}{1} \;\|\; \overline{1} \vee 2,\; \overline{3} \vee 4,\; \overline{5} \vee \overline{6},\; 6 \vee \overline{5} \vee \overline{2} & \Longrightarrow & (\text{UnitProp}) \\
\color{red}{1}\; 2 \;\|\; \overline{1} \vee 2,\; \overline{3} \vee 4,\; \overline{5} \vee \overline{6},\; 6 \vee \overline{5} \vee \overline{2} & \Longrightarrow & (\text{Decide}) \\
\color{red}{1}\; 2\; \color{red}{3} \;\|\; \overline{1} \vee 2,\; \overline{3} \vee 4,\; \overline{5} \vee \overline{6},\; 6 \vee \overline{5} \vee \overline{2} & &
\end{array}$$

Basic DPLL System – Example

$$
\begin{array}{rcll}
\emptyset & \| & \overline{1} \lor 2,\ \ \overline{3} \lor 4,\ \ \overline{5} \lor \overline{6},\ \ 6 \lor \overline{5} \lor \overline{2} & \Longrightarrow \ \ (\text{Decide}) \\
{\color{red}1} & \| & \overline{1} \lor 2,\ \ \overline{3} \lor 4,\ \ \overline{5} \lor \overline{6},\ \ 6 \lor \overline{5} \lor \overline{2} & \Longrightarrow \ \ (\text{UnitProp}) \\
{\color{red}1}\ 2 & \| & \overline{1} \lor 2,\ \ \overline{3} \lor 4,\ \ \overline{5} \lor \overline{6},\ \ 6 \lor \overline{5} \lor \overline{2} & \Longrightarrow \ \ (\text{Decide}) \\
{\color{red}1}\ 2\ {\color{red}3} & \| & \overline{1} \lor 2,\ \ \overline{3} \lor 4,\ \ \overline{5} \lor \overline{6},\ \ 6 \lor \overline{5} \lor \overline{2} & \Longrightarrow \ \ (\text{UnitProp}) \\
{\color{red}1}\ 2\ {\color{red}3}\ 4 & \| & \overline{1} \lor 2,\ \ \overline{3} \lor 4,\ \ \overline{5} \lor \overline{6},\ \ 6 \lor \overline{5} \lor \overline{2} &
\end{array}
$$

Basic DPLL System – Example

$$
\begin{array}{rcll}
\emptyset & \parallel & \overline{1} \vee 2,\ \ \overline{3} \vee 4,\ \ \overline{5} \vee \overline{6},\ \ 6 \vee \overline{5} \vee \overline{2} & \Longrightarrow \ (\text{Decide}) \\
{\color{red}1} & \parallel & \overline{1} \vee 2,\ \ \overline{3} \vee 4,\ \ \overline{5} \vee \overline{6},\ \ 6 \vee \overline{5} \vee \overline{2} & \Longrightarrow \ (\text{UnitProp}) \\
{\color{red}1}\ 2 & \parallel & \overline{1} \vee 2,\ \ \overline{3} \vee 4,\ \ \overline{5} \vee \overline{6},\ \ 6 \vee \overline{5} \vee \overline{2} & \Longrightarrow \ (\text{Decide}) \\
{\color{red}1}\ 2\ {\color{red}3} & \parallel & \overline{1} \vee 2,\ \ \overline{3} \vee 4,\ \ \overline{5} \vee \overline{6},\ \ 6 \vee \overline{5} \vee \overline{2} & \Longrightarrow \ (\text{UnitProp}) \\
{\color{red}1}\ 2\ {\color{red}3}\ 4 & \parallel & \overline{1} \vee 2,\ \ \overline{3} \vee 4,\ \ \overline{5} \vee \overline{6},\ \ 6 \vee \overline{5} \vee \overline{2} & \Longrightarrow \ (\text{Decide}) \\
{\color{red}1}\ 2\ {\color{red}3}\ 4\ {\color{red}5} & \parallel & \overline{1} \vee 2,\ \ \overline{3} \vee 4,\ \ \overline{5} \vee \overline{6},\ \ 6 \vee \overline{5} \vee \overline{2} &
\end{array}
$$

Basic DPLL System – Example

$$
\begin{array}{rcll}
\emptyset & \| & \overline{1} \lor 2,\ \ \overline{3} \lor 4,\ \ \overline{5} \lor \overline{6},\ \ 6 \lor \overline{5} \lor \overline{2} & \Longrightarrow \ (\text{Decide}) \\
1 & \| & \overline{1} \lor 2,\ \ \overline{3} \lor 4,\ \ \overline{5} \lor \overline{6},\ \ 6 \lor \overline{5} \lor \overline{2} & \Longrightarrow \ (\text{UnitProp}) \\
1\ 2 & \| & \overline{1} \lor 2,\ \ \overline{3} \lor 4,\ \ \overline{5} \lor \overline{6},\ \ 6 \lor \overline{5} \lor \overline{2} & \Longrightarrow \ (\text{Decide}) \\
1\ 2\ 3 & \| & \overline{1} \lor 2,\ \ \overline{3} \lor 4,\ \ \overline{5} \lor \overline{6},\ \ 6 \lor \overline{5} \lor \overline{2} & \Longrightarrow \ (\text{UnitProp}) \\
1\ 2\ 3\ 4 & \| & \overline{1} \lor 2,\ \ \overline{3} \lor 4,\ \ \overline{5} \lor \overline{6},\ \ 6 \lor \overline{5} \lor \overline{2} & \Longrightarrow \ (\text{Decide}) \\
1\ 2\ 3\ 4\ 5 & \| & \overline{1} \lor 2,\ \ \overline{3} \lor 4,\ \ \overline{5} \lor \overline{6},\ \ 6 \lor \overline{5} \lor \overline{2} & \Longrightarrow \ (\text{UnitProp}) \\
1\ 2\ 3\ 4\ 5\ \overline{6} & \| & \overline{1} \lor 2,\ \ \overline{3} \lor 4,\ \ \overline{5} \lor \overline{6},\ \ 6 \lor \overline{5} \lor \overline{2} &
\end{array}
$$

Basic DPLL System – Example

$$\begin{array}{rcll}
\emptyset & \| & \overline{1} \lor 2,\ \overline{3} \lor 4,\ \overline{5} \lor \overline{6},\ 6 \lor \overline{5} \lor \overline{2} & \Longrightarrow \text{(Decide)} \\
{\color{red}1} & \| & \overline{1} \lor 2,\ \overline{3} \lor 4,\ \overline{5} \lor \overline{6},\ 6 \lor \overline{5} \lor \overline{2} & \Longrightarrow \text{(UnitProp)} \\
{\color{red}1}\ 2 & \| & \overline{1} \lor 2,\ \overline{3} \lor 4,\ \overline{5} \lor \overline{6},\ 6 \lor \overline{5} \lor \overline{2} & \Longrightarrow \text{(Decide)} \\
{\color{red}1}\ 2\ {\color{red}3} & \| & \overline{1} \lor 2,\ \overline{3} \lor 4,\ \overline{5} \lor \overline{6},\ 6 \lor \overline{5} \lor \overline{2} & \Longrightarrow \text{(UnitProp)} \\
{\color{red}1}\ 2\ {\color{red}3}\ 4 & \| & \overline{1} \lor 2,\ \overline{3} \lor 4,\ \overline{5} \lor \overline{6},\ 6 \lor \overline{5} \lor \overline{2} & \Longrightarrow \text{(Decide)} \\
{\color{red}1}\ 2\ {\color{red}3}\ 4\ {\color{red}5} & \| & \overline{1} \lor 2,\ \overline{3} \lor 4,\ \overline{5} \lor \overline{6},\ 6 \lor \overline{5} \lor \overline{2} & \Longrightarrow \text{(UnitProp)} \\
{\color{red}1}\ 2\ {\color{red}3}\ 4\ {\color{red}5}\ \overline{6} & \| & \overline{1} \lor 2,\ \overline{3} \lor 4,\ \overline{5} \lor \overline{6},\ {\color{red}6 \lor \overline{5} \lor \overline{2}} & \Longrightarrow \text{(Backjump)} \\
{\color{red}1}\ 2\ \overline{5} & \| & \overline{1} \lor 2,\ \overline{3} \lor 4,\ \overline{5} \lor \overline{6},\ 6 \lor \overline{5} \lor \overline{2} &
\end{array}$$

Backjumpwith clause ${\color{red}\overline{1} \lor \overline{5}}$

Basic DPLL System – Example

$$\ldots$$

$$1\ 2\ 3\ 4\ 5\ \overline{6}\ \parallel\ \overline{1} \vee 2,\ \overline{3} \vee 4,\ \overline{5} \vee \overline{6},\ 6 \vee \overline{5} \vee \overline{2} \quad \Longrightarrow \quad (\text{Backjump})$$

$$1\ 2\ \overline{5}\ \parallel\ \overline{1} \vee 2,\ \overline{3} \vee 4,\ \overline{5} \vee \overline{6},\ 6 \vee \overline{5} \vee \overline{2}.$$

Basic DPLL System – Example

$$\ldots$$

$$1\ 2\ 3\ 4\ 5\ \overline{6}\ \parallel\ \ \overline{1} \vee 2,\ \ \overline{3} \vee 4,\ \ \overline{5} \vee \overline{6},\ \ 6 \vee \overline{5} \vee \overline{2} \quad \Longrightarrow \quad (\text{Backjump})$$

$$1\ 2\ \overline{5}\ \parallel\ \ \overline{1} \vee 2,\ \ \overline{3} \vee 4,\ \ \overline{5} \vee \overline{6},\ \ 6 \vee \overline{5} \vee \overline{2}.$$

$\overline{1} \vee \overline{5}$ is implied by the original set of clauses. For instance, by resolution,

$$\dfrac{\dfrac{\overline{1} \vee 2 \quad 6 \vee \overline{5} \vee \overline{2}}{\overline{1} \vee 6 \vee \overline{5}} \qquad \overline{5} \vee \overline{6}}{\overline{1} \vee \overline{5}}$$

Therefore, instead *deciding* 3, we could have *deduced* $\overline{5}$.

Basic DPLL System – Example

$$\ldots$$

$$1\ 2\ 3\ 4\ 5\ \overline{6} \parallel \quad \overline{1} \vee 2,\ \ \overline{3} \vee 4,\ \ \overline{5} \vee \overline{6},\ \ 6 \vee \overline{5} \vee \overline{2} \quad \Longrightarrow \quad (\text{Backjump})$$

$$1\ 2\ \overline{5} \parallel \quad \overline{1} \vee 2,\ \ \overline{3} \vee 4,\ \ \overline{5} \vee \overline{6},\ \ 6 \vee \overline{5} \vee \overline{2}.$$

$\overline{1} \vee \overline{5}$ is implied by the original set of clauses. For instance, by resolution,

$$\cfrac{\cfrac{\overline{1} \vee 2 \quad 6 \vee \overline{5} \vee \overline{2}}{\overline{1} \vee 6 \vee \overline{5}} \qquad \overline{5} \vee \overline{6}}{\overline{1} \vee \overline{5}}$$

Therefore, instead *deciding* 3, we could have *deduced* $\overline{5}$.

Clauses like $\overline{1} \vee \overline{5}$ are computed by navigating the *implication graph*.

The Eager Approach

- Translate formula into equisatisfiable propositional formula and use off-the-shelf SAT solver.
- Why "eager"?

  Search uses *all* theory information from the beginning.
- Can use best available SAT solver.
- Sophisticated encodings are need for each theory.
- Sometimes translation and/or solving too slow.

Lazy approach: SAT solvers + Theories

- This approach was independently developed by several groups: CVC (Stanford), ICS (SRI), MathSAT (Univ. Trento, Italy), and Verifun (HP).
- It was motivated also by the breakthroughs in SAT solving.
- SAT solver "manages" the boolean structure, and assigns truth values to the atoms in a formula.
- Efficient theory solvers is used to validate the (partial) assignment produced by the SAT solver.
- When theory solver detects unsatisfiability $\longrightarrow$ a new clause (*lemma*) is created.

SAT solvers + Theories (cont.)

- Example:
  - Suppose the SAT solver assigns $\{x = y \to \mathsf{T}, y = z \to \mathsf{T}, f(x) = f(z) \to \mathsf{F}\}$.
  - Theory solver detects the conflict, and a *lemma* is created $\neg(x = y) \vee \neg(y = z) \vee f(x) = f(z)$.
- Some theory solvers use the "proof" of the conflict to build the lemma.
- Problems in these tools:
  - The *lemmas are imprecise* (not minimal).
  - The theory solver is "passive": *it just detects conflicts*. There is no propagation step.
  - *Backtracking is expensive*, some tools restart from scratch when a conflict is detected.

Precise Lemmas

- Lemma:
  $\{a_1 = \textsf{T}, a_1 = \textsf{F}, a_3 = \textsf{F}\}$*is inconsistent* $\rightsquigarrow \neg a_1 \vee a_2 \vee a_3$
- An inconsistent A set is *redundant* if $A' \subset A$ is also inconsistent.
- Redundant inconsistent sets $\rightsquigarrow$ Imprecise Lemmas $\rightsquigarrow$ Ineffective pruning of the search space.
- *Noise* of a redundant set: $A \setminus A_{min}$.
- The imprecise lemma is *useless* in any context (partial assignment) where an atom in the noise has a different assignment.
- Example: suppose a_1 is in the noise, then $\neg a_1 \vee a_2 \vee a_3$ is useless when $a_1 = F$.

Theory Propagation

- The SAT solver is assigning truth values to the atoms in a formula.
- The partial assignment produced by the SAT solver may imply the truth value of unassigned atoms.
- Example:

$$x = y \wedge y = z \wedge (f(x) \neq f(z) \vee f(x) = f(w))$$

  The partial assignment $\{x = y \longrightarrow \mathsf{T}, y = z \longrightarrow \mathsf{T}\}$ implies $f(x) = f(z)$.

- Reduces the number of conflicts and the search space.

Efficient Backtracking

- One of the most important improvements in SAT was efficient backtracking.
- Until recently, backtracking was ignored in the design of theory solvers.
- Extreme (inefficient) approach: restart from scratch on every conflict.
- Other easy (and inefficient solutions):
  - Functional data-structures.
  - Backtrackable data-structures (trail-stack).
- *Backtracking should be included in the design of theory solvers.*
- *Restore to a logically equivalent state.*

The ideal theory solver

- Efficient in real benchmarks.
- Produces precise lemmas.
- Supports Theory Propagation.
- Incremental.
- Efficient Backtracking.
- Produces counterexamples.

Roadmap

- Background
- Theories
- Combination of Theories
- SAT + Theories
- Decision Procedures for Specific Theories
- Applications

Congruence Closure

$T_{\mathcal{E}}$-satisfiability can be decided with a simple algorithm known as congruence closure

Let $G = (V, E)$ be a directed graph such that for each vertex v in G, the successors of v are ordered.

Let C be any equivalence relation on V.

The *congruence closure* C^* of C is the finest equivalence relation on V that contains C and satisfies the following property for all vertices v and w:

> Let v and w have successors $v_1, \ldots, v_k$ and $w_1, \ldots, w_l$ respectively. If $k = l$ and $(v_i, w_i) \in C^*$ for $1 \leq i \leq k$, then $(v, w) \in C^*$.

Congruence Closure

Often, the vertices are labeled by some labeling function λ. In this case, the property becomes:

If $\lambda(v) = \lambda(w)$ and if $k = l$ and $(v_i, w_i) \in C^*$ for $1 \leq i \leq k$, then $(v, w) \in C^*$.

A Simple Algorithm

Let $C_0 = C$ and $i = 0$.

1. Number the equivalence classes in C_i.
2. Let α assign to each vertex v the number $\alpha(v)$ of the equivalence class containing v.
3. For each vertex v construct a *signature* $s(v) = \lambda(v)(\alpha(v_1), \ldots, \alpha(v_k))$, where $v_1, \ldots, v_k$ are the successors of v.
4. Group the vertices into equivalence classes by signature.
5. Let C_{i+1} be the finest equivalence relation on V such that two vertices equivalent under C_i or having the same signature are equivalent under C_{i+1}.
6. If $C_{i+1} = C_i$, let $C^* = C_i$; otherwise increment i and repeat.

Congruence Closure and $\mathcal{T}_{\mathcal{E}}$

Recall that $\mathcal{T}_{\mathcal{E}}$ is the empty theory with equality over some signature $\Sigma(C)$ containing only function symbols.

If Γ is a set of ground Σ-equalities and Δ is a set of ground $\Sigma(C)$-disequalities, then the satisfiability of $\Gamma \cup \Delta$ can be determined as follows.

- Let G be a graph which corresponds to the abstract syntax trees of terms in $\Gamma \cup \Delta$, and let v_t denote the vertex of G associated with the term t.
- Let C be the equivalence relation on the vertices of G induced by Γ.
- $\Gamma \cup \Delta$ is satisfiable iff for each $s \neq t \in \Delta$, $(v_s, v_t) \notin C^*$.

Difference Logic

- Graph interpretation:
  - Variables are nodes.
  - Atoms $x - y \leq c$ are weighted edges: $y \xrightarrow{c} x$.
  - A set of literals is satisfiable iff there is no negative cycle: $x_1 \xrightarrow{c_1} x_2 \ldots x_n \xrightarrow{c_n} x_1$, $C = c_1 + \ldots + c_n < 0$. That is, negative cycle implies $0 \leq C < 0$.
  - Bellman-Ford like algorithm to find such cycles in $O(mn)$.

Linear arithmetic

- Most SMT solvers use algorithms based on Fourier-Motzkin or Simplex.
- Fourier Motzkin:
  - Variable elimination method.
  - $t_1 \leq ax,\ bx \leq t_2 \leadsto bt_1 \leq at_2$
  - Polynomial time for difference logic.
  - Double exponential and consumes a lot of memory.
- Simplex:
  - Very efficient in practice.
  - Worst-case exponential (I've never seen this behavior in real benchmarks).

Fast Linear Arithmetic

- Simplex General Form.
- New algorithm based on the Dual Simplex.
- Efficient Backtracking.
- Efficient Theory Propagation.
- New approach for solving strict inequalities ($t > 0$).
- Preprocessing step.
- It *outperforms* even solvers using algorithms for the *Difference Logic* fragment.

Fast Linear Arithmetic: General Form

- *General Form:* $Ax = 0$ and $l_j \leq x_j \leq u_j$
- Example:

$$x \geq 0 \land (x + y \leq 2 \lor x + 2y \geq 6) \land (x + y = 2 \lor x + 2y > 4)$$

$$\leadsto$$

$$(s_1 = x + y \land s_2 = x + 2y) \ \land$$
$$(x \geq 0 \land (s_1 \leq 2 \lor s_2 \geq 6) \land (s_1 = 2 \lor s_2 > 4))$$

- Only *bounds* (e.g., $s_1 \leq 2$) are asserted during the search.
- *Unconstrained variables* can be *eliminated* before the beginning of the search.

Equations + Bounds + Assignment

- An *assignment* β is a mapping from variables to values.
- We maintain an *assignment* that satisfies all *equations* and *bounds*.
- The assignment of non basic variables implies the assignment of basic variables.
- *Equations + Bounds* can be used to derive *new bounds*.
- Example: $x = y - z,\ y \leq 2,\ z \geq 3 \leadsto x \leq -1$.
- The *new bound* may be inconsistent with the already known bounds.
- Example: $x \leq -1,\ x \geq 0$.

Roadmap

- Background
- Theories
- Combination of Theories
- SAT + Theories
- Decision Procedures for Specific Theories
- Applications

Bounded Model Checking (BMC)

- To check whether a program with initial state I and next-state relation T violates the invariant *Inv* in the first k steps, one checks:

$$I(s_0) \wedge T(s_0, s_1) \wedge \ldots \wedge T(s_{k-1}, s_k) \wedge (\neg \mathit{Inv}(s_0) \vee \ldots \vee \neg \mathit{Inv}(s_k))$$

- This formula is satisfiable if and only if there exists a path of length at most k from the initial state s_0 which violates the invariant k.
- Formulas produced in BMC are usually quite big.
- The SAL bounded model checker from SRI uses SMT solvers.
`http://sal.csl.sri.com`

MUTT: MSIL Unit Testing Tools

- `http://research.microsoft.com/projects/mutt`
- *Unit tests are popular*, but it is far from trivial to write them.
- It is quite laborious to write enough of them to have confidence in the correctness of an implementation.
- Approach: *symbolic execution*.
- Symbolic execution builds a path condition over the input symbols.
- A *path condition* is a mathematical formula that encodes data constraints that result from executing a given code path.

MUTT: MSIL Unit Testing Tools

- When symbolic execution reaches a if-statement, it will explore two execution paths:
  1. The if-condition is conjoined to the path condition for the then-path.
  2. The negated condition to the path condition of the else-path.
- SMT solver must be able to produce models.
- SMT solver is also used to test path *feasibility*.

Spec#: Extended Static Checking

- `http://research.microsoft.com/specsharp/`
- Superset of C#
  - non-null types
  - pre- and postconditions
  - object invariants
- Static program verification
- Example:

```
public StringBuilder Append(char[] value, int startIndex,
                            int charCount);
 requires value == null ==> startIndex == 0 && charCount == 0;
 requires 0 <= startIndex;
 requires 0 <= charCount;
 requires value == null ||
          startIndex + charCount <= value.Length;
```

Spec#: Architecture

- Verification condition generation:

  Spec# compiler: Spec# $\rightsquigarrow$ MSIL (bytecode).

  Bytecode translator: MSIL $\rightsquigarrow$ Boogie PL.

  V.C. generator: Boogie PL $\rightsquigarrow$ SMT formula.

- SMT solver is used to prove the verification conditions.
- Counterexamples are traced back to the source code.
- *The formulas produces by Spec# are not quantifier free.*
  - Heuristic quantifier instantiation is used.

SLAM: device driver verification

- `http://research.microsoft.com/slam/`
- *SLAM/SDV* is a software model checker.
- Application domain: *device drivers*.
- Architecture

  c2bp C program $\rightsquigarrow$ boolean program (*predicate abstraction*).

  bebop Model checker for boolean programs.

  newton Model refinement (*check for path feasibility*)
- SMT solvers are used to perform predicate abstraction and to check path feasibility.
- c2bp makes several calls to the SMT solver. The formulas are relatively small.

Conclusion

- SMT is the next generation of verification engines.
- More automation: it is push-button technology.
- SMT solvers are used in different applications.
- The breakthrough in SAT solving influenced the new generation of SMT solvers:
  - Precise lemmas.
  - Theory Propagation.
  - Incrementality.
  - Efficient Backtracking.

References

[Ack54] W. Ackermann. Solvable cases of the decision problem. *Studies in Logic and the Foundation of Mathematics*, 1954

[ABC$^+$02] G. Audemard, P. Bertoli, A. Cimatti, A. Kornilowicz, and R. Sebastiani. A SAT based approach for solving formulas over boolean and linear mathematical propositions. In *Proc. of CADE'02*, 2002

[BDS00] C. Barrett, D. Dill, and A. Stump. A framework for cooperating decision procedures. In *17th International Conference on Computer-Aided Deduction*, volume 1831 of *Lecture Notes in Artificial Intelligence*, pages 79–97. Springer-Verlag, 2000

[BdMS05] C. Barrett, L. de Moura, and A. Stump. SMT-COMP: Satisfiability Modulo Theories Competition. In *Int. Conference on Computer Aided Verification (CAV'05)*, pages 20–23. Springer, 2005

[BDS02] C. Barrett, D. Dill, and A. Stump. Checking satisfiability of first-order formulas by incremental translation to SAT. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Proceedings of the* 14^{th} *International Conference on Computer Aided Verification (CAV '02)*, volume 2404 of *Lecture Notes in Computer Science*, pages 236–249. Springer-Verlag, July 2002. Copenhagen, Denmark

[BBC$^+$05] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P. van Rossum, S. Ranise, and R. Sebastiani. Efficient satisfiability modulo theories via delayed theory combination. In *Int. Conf. on Computer-Aided Verification (CAV)*, volume 3576 of *LNCS*. Springer, 2005

[Chv83] V. Chvatal. *Linear Programming*. W. H. Freeman, 1983

References

[CG96] B. Cherkassky and A. Goldberg. Negative-cycle detection algorithms. In *European Symposium on Algorithms*, pages 349–363, 1996

[DLL62] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, July 1962

[DNS03] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Labs, 2003

[DST80] P. J. Downey, R. Sethi, and R. E. Tarjan. Variations on the Common Subexpression Problem. *Journal of the Association for Computing Machinery*, 27(4):758–771, 1980

[dMR02] L. de Moura and H. Rueß. Lemmas on demand for satisfiability solvers. In *Proceedings of the Fifth International Symposium on the Theory and Applications of Satisfiability Testing (SAT 2002)*. Cincinnati, Ohio, 2002

[DdM06] B. Dutertre and L. de Moura. Integrating simplex with DPLL(T). Technical report, CSL, SRI International, 2006

[GHN$^+$04] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast decision procedures. In R. Alur and D. Peled, editors, *Int. Conference on Computer Aided Verification (CAV 04)*, volume 3114 of *LNCS*, pages 175–188. Springer, 2004

References

[MSS96] J. Marques-Silva and K. A. Sakallah. GRASP - A New Search Algorithm for Satisfiability. In *Proc. of ICCAD'96*, 1996

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

[NO05] R. Nieuwenhuis and A. Oliveras. DPLL(T) with exhaustive theory propagation and its application to difference logic. In *Int. Conference on Computer Aided Verification (CAV'05)*, pages 321–334. Springer, 2005

[Opp80] D. Oppen. Reasoning about recursively defined data structures. *J. ACM*, 27(3):403–411, 1980

[PRSS99] A. Pnueli, Y. Rodeh, O. Shtrichman, and M. Siegel. Deciding equality formulas by small domains instantiations. *Lecture Notes in Computer Science*, 1633:455–469, 1999

[Pug92] William Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. In *Communications of the ACM*, volume 8, pages 102–114, August 1992

[RT03] S. Ranise and C. Tinelli. The smt-lib format: An initial proposal. In *Proceedings of the 1st International Workshop on Pragmatics of Decision Procedures in Automated Reasoning (PDPAR'03), Miami, Florida*, pages 94–111, 2003

References

[RS01] H. Ruess and N. Shankar. Deconstructing shostak. In *16th Annual IEEE Symposium on Logic in Computer Science*, pages 19–28, June 2001

[SLB03] S. Seshia, S. Lahiri, and R. Bryant. A hybrid SAT-based decision procedure for separation logic with uninterpreted functions. In *Proc. 40th Design Automation Conference*, pages 425–430. ACM Press, 2003

[Sho81] R. Shostak. Deciding linear inequalities by computing loop residues. *Journal of the ACM*, 28(4):769–779, October 1981