SMT Solvers: Introduction & Applications

Cambridge 2007

Leonardo de Moura

leonardo@microsoft.com

Microsoft Research

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.

Examples

- $x + 2 = y \Rightarrow f(\textit{read}(\textit{write}(a, x, 3), y - 2)) = f(y - x + 1)$
- $f(f(x) - f(y)) \neq f(z), x + z \leq y \leq x \Rightarrow z < 0$

Applications

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

SMT-Solvers & SMT-Lib & SMT-Comp

- SMT-Solves:

  ArgoLib, Ario, Barcelogic, CVC, CVC Lite, CVC3, ExtSAT, Fx7, Harvey, HTP, *ICS (SRI)*, Jat, MathSAT, Sateen, Simplify, Spear, STeP, STP, SVC, TSAT, UCLID, *Yices (SRI)*, Zap (Microsoft), *Z3 (Microsoft)*

- SMT-Lib: library of benchmarks

  `http://www.smtlib.org/`

- SMT-Comp: annual SMT-Solver competition

  `http://www.smtcomp.org/`

Roadmap

- Background
- Theories
- Combination of Theories
- Quantifiers
- Applications

Language

- A *signature* Σ is a finite set of: function symbols Σ_F = $\{f, g, \ldots\}$, predicate symbols Σ_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* $\{x, y, \ldots\}$ disjoint of Σ.
- *Terms:*

  $$t \ := \ f(t_1, \ldots, t_n) \mid x$$

- *Formulas:*

  $$\phi := p(t_1, \ldots, t_n) \mid \phi_1 \vee \phi_2 \mid \phi_1 \wedge \phi_2 \mid \neg\phi_1 \mid \exists x : \phi_1 \mid \forall x : \phi_1$$

- *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*.

Roadmap

- Background
- Theories
- Combination of Theories
- Quantifiers
- 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.

Theory of Bit-vectors

- Bit-vectors (also called “words”) are a crucial abstraction for reasoning about hardware and software structures.
- Operations:
  - Concatenation.
  - Extraction.
  - Bit-wise operations.
  - Modular arithmetic.
- Deciding equality of arbitrary combinations of these operations is NP-hard.

Transitive closure

- The *transitive closure* R^* of a binary relation R on a set X is the smallest transitive relation on X that contains R.
- Useful for reasoning about programs that manipulate heap objects (e.g., lists and trees).
- For example, if X is a set of heap objects, and $R(x, y)$ is the relation 'x points to y', then $R^*(x, y)$ is the relation 'x reaches y'.

Other theories

- Partial orders
- Tuples & Records
- Inductive datatypes
- . . .

Roadmap

- Background
- Theories
- Combination of Theories
- Quantifiers
- 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. $\mathcal{T}_2 = DC(\forall x, y, z.\ (x = y) \vee (x = z) \vee (y = z))$.
- 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”).

$$\begin{aligned}\mathcal{T}_2{}' = DC(\forall x, y, z.\ in_2(x) \wedge in_2(y) \wedge in_2(z) \Rightarrow \\ (x = y) \vee (x = z) \vee (y = z))\end{aligned}$$

- *$\mathcal{T}_2{}'$ 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.

$$y = 1, z = 2, 1 \leq x \leq 2 \models x = y \vee x = z$$

  - Nonlinear arithmetic.

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

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

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

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 $O(n^3 \times (T_1(n) + T_2(n)))$.

Reduction Functions

- A *reduction function* reduces the satisfiability problem for a theory $\mathcal{T}_1$ to the satisfiability problem of a simpler theory $\mathcal{T}_2$.
- Reduction functions simplify the implementation.
- Potential disadvantages:
  - “Information loss”.
  - Eager addition of irrelevant information.
- 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, add the equality $f(a, b) = f(b, a)$.

Reduction Functions: Ackermann's reduction

- *Ackermann's 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 clause $\vec{a} \neq \vec{c} \vee f_{\vec{a}} = f_{\vec{c}}$.
  - Replace $f(\vec{a})$ with $f_{\vec{a}}$ in ϕ.
- It is used in some SMT solvers to reduce $\mathcal{T}_{\mathcal{LA}} \cup \mathcal{T}_{\mathcal{E}}$ to $\mathcal{T}_{\mathcal{LA}}$.
- *Main problem: quadratic number of new clauses.*
- It is also problematic to use this approach in the context of *several theories* and when combining SMT solvers with *quantifier instantiation*.

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 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.

Roadmap

- Background
- Theories
- Combination of Theories
- Quantifiers
- Applications

Quantifiers

- Since first-order logic is undecidable, satisfiability is not solvable for arbitrary quantified formulas.
- Some theories, e.g., datatypes, linear arithmetic over integers, arithmetic over reals, support quantifier elimination.
- Existential quantifiers can be skolemized, but the problem of instantiating universal quantifiers for detecting unsatisfiability remains.

Heuristic Quantifier Instantiation

- Semantically, $\forall x_1, \ldots, x_n . F$ is equivalent to the infinite conjunction $\bigwedge_\beta \beta(F)$.
- Solvers use heuristics to select from this infinite conjunction those instances that are "relevant".
- The key idea is to treat an instance $\beta(F)$ as relevant whenever it contains enough terms that are represented in the solver state.
- Non ground terms p from F are selected as *patterns*.
- *E-matching* (matching modulo equalities) is used to find instances of the patterns.
- Example: $f(a, b)$ matches the pattern $f(g(x), x)$ if $a = g(b)$.

E-matching

- E-matching is NP-hard.
- The number of matches can be exponential.
- It is not refutationally complete.
- In practice:
  - Indexing techniques for fast retrieval.
  - Incremental E-matching.

E-matching: example

- $\forall x. f(g(x)) = x$
- Pattern: $f(g(x))$
- Atoms: $a = g(b), b = c, f(a) \neq c$
- $\longrightarrow$ *instantiate* $f(g(b)) = b$

E-matching limitations

- E-matching needs ground (seed) terms.
  - It fails to prove simple properties when ground (seed) terms are not available.
  - Example:

$$(\forall x. f(x) \leq 0) \wedge (\forall x. f(x) > 0)$$

- Matching loops

$$(\forall x. f(x) = g(f(x))) \wedge (\forall x. g(x) = f(g(x)))$$

  - Inefficiency and/or non-termination.
  - Some solvers have support for detecting matching loops based on instantiation chain length.

Quantifiers: future work

- When quantifiers are used, solvers such as Yices and Z3 produce "candidate models".
- Model checking.

  Evaluate quantifiers using the "candidate model", use violations to create new quantifier instantiations.
- Superposition calculus + SMT.
- Support for decidable fragments.

Roadmap

- Background
- Theories
- Combination of Theories
- Quantifiers
- 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 I.
- 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 $\leadsto$ 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.

Interactive Theorem Provers

- *SMT solvers can be used inside of an interactive theorem provers.*
- More automation.
- In PVS, goals can be discharged using Yices.
- Model generation: improved “quick-check”.

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 pair $\langle M, \delta \rangle$ for machine M and duration δ.
- Find a schedule with a minimum duration.
- Incremental SMT solving: binary search in $[0, \Sigma_i \delta_i]$.
- Example:

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 source city and 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 predicates: *location*(t, c, i), *at*(p, c, i), *on*(p, t, i).
- 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.

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.