universität
innsbruck

SAT and SMT Solving

Sarah Winkler

Computational Logic Group
Department of Computer Science
University of Innsbruck

lecture 5
SS 2019

Outline

- Summary of Last Week
- Satisfiability Modulo Theories
- DPLL(T)
- Equality and Uninterpreted Functions in Practice

Definitions

for unsatisfiable CNF formula φ given as set of clauses

- $\psi \subseteq \varphi$ such that $\bigwedge_{C \in \psi} C$ is unsatisfiable is unsatisfiable core (UC) of φ
- minimal unsatisfiable core ψ is UC such that every subset of ψ is satisfiable
- SUC (minimum unsatisfiable core) is UC such that $|\psi|$ is minimal

Remark

SUC is always minimal unsatisfiable core

Definition (Resolution Graph)

directed acyclic graph $G = (V, E)$ is resolution graph for set of clauses φ if

1. $V = V_i \uplus V_c$ is set of clauses and $V_i = \varphi$,
2. V_i nodes have no incoming edges,
3. there is exactly one node $\Box$ without outgoing edges,
4. $\forall C \in V_c \; \exists$ edges $D \to C$, $D' \to C$ such that C is resolvent of D and D', and
5. there are no other edges.

Algorithm minUnsatCore(φ)

Input: unsatisfiable formula φ
Output: minimal unsatisfiable core of φ

build resolution graph $G = (V_i \uplus V_c, E)$ for φ
while $\exists$ unmarked clause in V_i **do**
  $C \leftarrow$ unmarked clause in V_i
  if SAT($\overline{Reach_G(C)}$) **then** ▷ subgraph without C satisfiable?
    mark C ▷ C is UC member
  else
    build resolution graph $G' = (V'_i \uplus V'_c, E')$ for $\overline{Reach_G(C)}$
    $V_i \leftarrow V_i \setminus \{C\}$ and $V_c \leftarrow V'_c \cup (V_c \setminus Reach_G(C))$
    $E \leftarrow E' \cup (E \setminus Reach^E_G(C))$
    $G \leftarrow (V_i \cup V_c, E)$
    $G \leftarrow G|_{BReach_G(\Box)}$ ▷ restrict to nodes with path to $\Box$
return V_i

Theorem

if φ unsatisfiable then minUnsatCore(φ) is minimal unsatisfiable core of φ

Definition (Partial minUNSAT)

pminUNSAT(χ, φ) is minimal $|\psi|$ such that $\psi \subseteq \varphi$ and $\chi \wedge \bigwedge_{C \in \psi} \neg C$ satisfiable

Lemma

$$|\varphi| = |\text{pminUNSAT}(\chi, \varphi)| + |\text{pmaxSAT}(\chi, \varphi)|$$

Theorem

FuMalik(χ, φ) = pminUNSAT(χ, φ)

Algorithm FuMalik(χ, φ)

Input: clause set φ and satisfiable clause set χ
Output: minUNSAT(χ, φ)

```
cost ← 0
while ¬SAT(χ ∪ φ) do
    UC ← unsatCore(χ ∪ φ)
    B ← ∅
    for C ∈ UC ∩ φ do                          ▷ loop over soft clauses in core
        b ← new blocking variable
        φ ← φ \ {C} ∪ {C ∨ b}
        B ← B ∪ {b}
    χ ← χ ∪ CNF(∑_{b∈B} b = 1)                 ▷ cardinality constraint is hard
    cost ← cost + 1
return cost
```

Outline

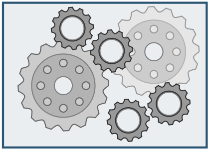
- Summary of Last Week
- Satisfiability Modulo Theories
- DPLL(T)
- Equality and Uninterpreted Functions in Practice

SMT Solving

input:	formula φ involving theory T	
output:	SAT + valuation v such that $v(\varphi) = T$	if φ satisfiable
	UNSAT	otherwise

φ $\longrightarrow$

$a + b \geqslant c \vee (a = 0 \wedge p)$



SMT solver

$\nearrow$ SAT (v) $v(a) = 3 \quad v(b) = 0$ $\quad v(c) = 0 \quad v(p) = \mathrm{T}$

$\searrow$ UNSAT

Example (Theories)

- arithmetic $\qquad 2a + b \geqslant c \vee (a - b = c + 3 \wedge p)$
- uninterpreted functions $\qquad \mathrm{f}(x, y) \neq \mathrm{f}(y, x) \wedge \mathrm{g}(\mathrm{a}) \rightarrow \mathrm{g}(\mathrm{f}(x, x)) = \mathrm{g}(y)$
- bit vectors $\qquad ((\mathrm{zext}_{32}\ a_8) + b_{32}) \times c_{32} >_u 0_{32}$

Definitions

for formulas F and G and list of literals M:

- theory T is set of first-order logic formulas without free variables
- F is T-consistent (or T-satisfiable) if $F \wedge T$ is satisfiable in first-order sense
- F is T-inconsistent (or T-unsatisfiable) if not T-consistent
- $M = l_1, \dots, l_k$ is T-consistent if $l_1 \wedge \cdots \wedge l_k$ is
- M is T-model of F if $M \vDash F$ and M is T-consistent
- F entails G in T (denoted $F \vDash_T G$) if $F \wedge \neg G$ is T-inconsistent
- F and G are T-equivalent (denoted $F \equiv_T G$) if $F \vDash_T G$ and $G \vDash_T F$

Definition (Theory of Equality)

theory of equality (EQ) uses binary predicate $\approx$ and consists of axioms

$$\forall x.\ (x \approx x) \qquad \forall x\, y.\ (x \approx y\ \rightarrow\ y \approx x) \qquad \forall x\, y\, z.\ (x \approx y \wedge y \approx z\ \rightarrow\ x \approx z)$$

Example

- $u \approx v \wedge \neg(v \approx w)$ is EQ-consistent
- $u \approx v \wedge \neg(v \approx w) \wedge (w \approx u \vee u \approx w)$ is EQ-inconsistent
- have $u \approx v \wedge \neg(v \approx w) \vDash_{\mathrm{EQ}} \neg(w \approx u)$ and $u \approx v \equiv_{\mathrm{EQ}} v \approx u$

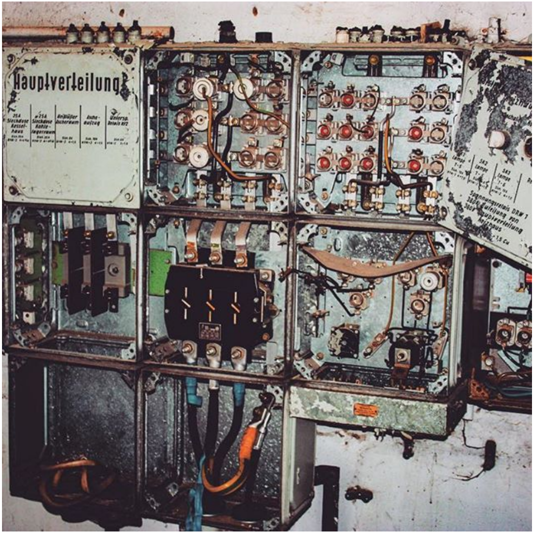
Definition (Theory of Equality With Uninterpreted Functions)
EUF over set of function symbols $\mathcal{F}$ consists of equality axioms:

$$\forall x.\ (x \approx x) \quad \forall x\ y.\ (x \approx y \ \rightarrow\ y \approx x) \quad \forall x\ y\ z.\ (x \approx y \wedge y \approx z \ \rightarrow\ x \approx z)$$

plus for all $f \in \mathcal{F}$ with n arguments the functional consistency axiom:

$$\forall x_1 y_1\ \dots\ x_n y_n\ (x_1 \approx y_1 \wedge \cdots \wedge x_n \approx y_n \ \rightarrow\ f(x_1, \dots, x_n) \approx f(y_1, \dots, y_n))$$

Uninterpreted Functions in Real Life



Definition (Theory of Equality With Uninterpreted Functions)

EUF over set of function symbols $\mathcal{F}$ consists of equality axioms:

$$\forall x\ (x \approx x) \qquad \forall x\, y\ (x \approx y \rightarrow y \approx x) \qquad \forall x\, y\, z\ (x \approx y \wedge y \approx z \rightarrow x \approx z)$$

plus for all $f \in \mathcal{F}$ with $n > 0$ arguments the functional consistency axiom:

$$\forall x_1 y_1 \ldots x_n y_n.\ (x_1 \approx y_1 \wedge \cdots \wedge x_n \approx y_n \rightarrow f(x_1, \ldots, x_n) \approx f(y_1, \ldots, y_n))$$

Example

EUF over $\mathcal{F} = \{\mathsf{a}/0,\ \mathsf{b}/0,\ \mathsf{f}/1,\ \mathsf{add}/2\}$ consists of axioms

$$\forall x\ (x \approx x) \qquad \forall x\, y\ (x \approx y \rightarrow y \approx x) \qquad \forall x\, y\, z\ (x \approx y \wedge y \approx z \rightarrow x \approx z)$$

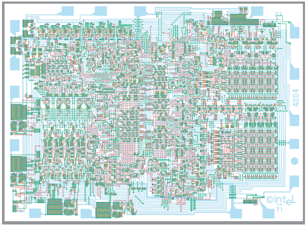
plus

$$\forall x\, y.\ (x \approx y \rightarrow \mathsf{f}(x) \approx \mathsf{f}(y))$$
$$\forall x_1\, y_1\, x_2\, y_2.\ (x_1 \approx y_1 \wedge x_2 \approx y_2 \rightarrow \mathsf{add}(x_1, y_1) \approx \mathsf{add}(x_2, y_2))$$

- $\mathsf{a} \not\approx \mathsf{b} \wedge \mathsf{f}(\mathsf{a}) \approx \mathsf{f}(\mathsf{b})$ is EUF-consistent
- $\mathsf{a} \not\approx y \wedge \mathsf{f}(\mathsf{a}) \approx x$ is EUF-consistent
- $\mathsf{a} \approx \mathsf{f}(\mathsf{b}) \wedge \mathsf{b} \approx \mathsf{f}(\mathsf{a}) \wedge \mathsf{f}(\mathsf{b}) \not\approx \mathsf{f}(\mathsf{f}(\mathsf{f}(\mathsf{b})))$ is EUF-inconsistent
- $\mathsf{a} \approx \mathsf{b} \vDash_{\mathsf{EUF}} \mathsf{f}(\mathsf{b}) \approx \mathsf{f}(\mathsf{a})$ but $\mathsf{a} \approx \mathsf{b} \nvDash_{\mathsf{EUF}} \mathsf{f}(\mathsf{b}) \approx \mathsf{f}(\mathsf{a})$

Application: Verification of Microprocessors

- verify that 3-stage pipelined MIPS processor satisfies intended instruction set architecture
- encoding
  - data as bit sequence
  - memory as uninterpreted function (UF)
  - computation logic as UF
  - control logic as uninterpreted predicate
- EUF ensures functional consistency: same data results in same computation



Miroslav N. Velev and Randal E. Bryant.
Bit-level abstraction in the verification of pipelined microprocessors by correspondence checking.
In Proc. of Formal Methods in Computer-Aided Design, pp. 18–35, 1998.

Theories of Interest in SMT Solvers

- equality + uninterpreted functions (EUF) $f(x,a) \approx g(y)$
- difference logic (DL) $x - y \leqslant 1$
- linear arithmetic $3x - 5y + 7z \leqslant 1$
  - over integers $\mathbb{Z}$ (LIA)
  - over reals $\mathbb{R}$ (LRA)
- arrays (A) $\mathsf{read}(\mathsf{write}(A,i,v),j)$
- bitvectors (BV) $((\mathsf{zext}_{32}\ a_8) + b_{32}) \times c_{32} >_u 0_{32}$
- strings $x \mathbin{@} y = z \mathbin{@} \mathsf{replace}(y, \mathtt{a}, \mathtt{b})$
- ...
- their combinations

SMT-LIB

- language standard and benchmarks: `http://www.smt-lib.org`
- annual solver competition: `http://www.smt-comp.org`
- solvers: Yices, OpenSMT, MathSAT, Z3, CVC4, Barcelogic, ...

The Eager Paradigm

Challenge

consider formula φ mixing propositional logic with theory T

Eager SMT Solving

- use satisfiability-preserving transformation from T literals to SAT formula, ship one big formula to SAT solver
- requires sophisticated translation for each theory:
  done for EUF, difference logic, linear integer arithmetic, arrays
- still dominant approach for bit-vector arithmetic (known as "bit blasting")
- advantage: use SAT solver off the shelf
- drawbacks:
  - expensive translations: infeasible for large formulas
  - even more complicated with multiple theories

The Lazy Paradigm

Challenge

consider formula φ mixing propositional logic with theory T

Idea

use specialized T-solver that can deal with conjunction of theory literals

Lazy SMT Solving

1. abstract φ to CNF:
   - "forget theory" by replacing T-literals with fresh propositional variables
   - obtain pure SAT formula, transform to CNF formula ψ
2. ship ψ to SAT solver
   - if ψ unsatisfiable, so is φ
   - if ψ satisfiable by v, check v with T-solver:
     - if v is T-consistent then also φ is satisfiable
     - otherwise T-solver generates T-consequence C of φ excluding v, repeat from 1 with $\varphi \wedge C$

Example

$$\mathsf{g}(\mathsf{a}) \approx \mathsf{c} \wedge (\neg(\mathsf{f}(\mathsf{g}(\mathsf{a})) \approx \mathsf{f}(\mathsf{c})) \vee \mathsf{g}(\mathsf{a}) \approx \mathsf{d}) \wedge \neg(\mathsf{c} \approx \mathsf{d})$$

- abstract to propositional skeleton $\psi_1 = x_1 \wedge (\neg x_2 \vee x_3) \wedge \neg x_4$
  satisfiable: $v_1(x_1) = \mathsf{T}$ and $v_1(x_2) = v_1(x_4) = \mathsf{F}$
- T-solver gets $\mathsf{g}(\mathsf{a}) \approx \mathsf{c} \wedge \mathsf{f}(\mathsf{g}(\mathsf{a})) \not\approx \mathsf{f}(\mathsf{c}) \wedge \mathsf{c} \not\approx \mathsf{d}$
  T-unsatisfiable: $\mathsf{g}(\mathsf{a}) \approx \mathsf{c}$ implies $\mathsf{f}(\mathsf{g}(\mathsf{a})) \approx \mathsf{f}(\mathsf{c})$
- block valuation v_1 in future: add $\neg x_1 \vee x_2 \vee x_4$

- $\psi_2 = x_1 \wedge (\neg x_2 \vee x_3) \wedge \neg x_4 \wedge (\neg x_1 \vee x_2 \vee x_4)$
  satisfiable: $v_2(x_1) = v_2(x_2) = v_2(x_3) = \mathsf{T}$ and $v_2(x_4) = \mathsf{F}$
- T-solver gets $\mathsf{g}(\mathsf{a}) \approx \mathsf{c} \wedge \mathsf{f}(\mathsf{g}(\mathsf{a})) \approx \mathsf{f}(\mathsf{c}) \wedge \mathsf{g}(\mathsf{a}) \approx \mathsf{d} \wedge \mathsf{c} \not\approx \mathsf{d}$
  T-unsatisfiable
- block valuation v_2 in future: add $\neg x_1 \vee \neg x_2 \vee \neg x_3 \vee x_4$

- $\psi_3 = x_1 \wedge (\neg x_2 \vee x_3) \wedge \neg x_4 \wedge (\neg x_1 \vee x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3 \vee x_4)$
- unsatisfiable

Outline

- Summary of Last Week
- Satisfiability Modulo Theories
- DPLL(T)
- Equality and Uninterpreted Functions in Practice

Approach

- most state-of-the-art SMT solvers use DPLL(T):
  lazy approach combining DPLL with theory propagation
- advantages: not specific to theory, also extends to theory combinations

Definition (DPLL(T) Transition Rules)

DPLL(T) consists of DPLL rules unit propagate, decide, fail, and restart plus

- T-backjump $\qquad M\, l^d\, N \parallel F, C \quad \Longrightarrow \quad M\, l' \parallel F, C$
  if $M\, l^d\, N \vDash \neg C$ and $\exists$ clause $C' \vee l'$ such that
  - $F, C \vDash_T C' \vee l'$
  - $M \vDash \neg C'$ and l' is undefined in M, and l' or l'^c occurs in F or in $M\, l^d\, N$
- T-learn $\qquad M \parallel F \quad \Longrightarrow \quad M \parallel F,\ C$
  if $F \vDash_T C$ and all atoms of C occur in M or F
- T-forget $\qquad M \parallel F,\ C \quad \Longrightarrow \quad M \parallel F$
  if $F \vDash_T C$
- T-propagate $\qquad M \parallel F \quad \Longrightarrow \quad M\, l \parallel F$
  if $M \vDash_T l$, literal l or l^c occurs in F, and l is undefined in M

Naive Lazy Approach in DPLL(T)

- whenever state $M \parallel F$ is final wrt unit propagate, decide, fail, T-backjump: check T-consistency of M with T-solver
- if M is T-consistent then satisfiability is proven
- otherwise $\exists l_1, \ldots, l_k$ subset of M such that $F \vDash_T \neg(l_1 \wedge \cdots \wedge l_k)$
- use T-learn to add $\neg l_1 \vee \cdots \vee \neg l_k$
- apply restart

Improvement 1: Incremental T-Solver

- T-solver checks T-consistency of model M whenever literal is added to M

Improvement 2: On-Line SAT solver

- after T-learn added clause, apply fail or T-backjump instead of restart

Improvement 3: Eager Theory Propagation

- apply T-propagate before decide

Remark

all three improvements can be combined

Example (Revisited with DPLL(T))

$$\underbrace{\mathsf{g(a)} \approx \mathsf{c}}_{1} \wedge (\neg\underbrace{\mathsf{f(g(a))} \approx \mathsf{f(c)}}_{2}) \vee \underbrace{\mathsf{g(a)} \approx \mathsf{d}}_{3}) \wedge \neg(\underbrace{\mathsf{c} \approx \mathsf{d}}_{4})$$

	$\parallel 1,\ (\overline{2} \vee 3),\ \overline{4}$	
$\Longrightarrow$	$1 \parallel 1,\ (\overline{2} \vee 3),\ \overline{4}$	unit propagate
$\Longrightarrow$	$1\,\overline{4} \parallel 1,\ (\overline{2} \vee 3),\ \overline{4}$	unit propagate
$\Longrightarrow$	$1\,\overline{4}\,\overline{2}^d \parallel 1,\ (\overline{2} \vee 3),\ \overline{4}$	decide
$\Longrightarrow$	$1\,\overline{4}\,\overline{2}^d \parallel 1,\ (\overline{2} \vee 3),\ \overline{4},\ (\overline{1} \vee 2 \vee 4)$	T-learn
$\Longrightarrow$	$1\,\overline{4}\,2 \parallel 1,\ (\overline{2} \vee 3),\ \overline{4},\ (\overline{1} \vee 2 \vee 4)$	T-backjump
$\Longrightarrow$	$1\,\overline{4}\,2\,3 \parallel 1,\ (\overline{2} \vee 3),\ \overline{4},\ (\overline{1} \vee 2 \vee 4)$	unit propagate
$\Longrightarrow$	$1\,\overline{4}\,2\,3 \parallel 1,\ (\overline{2} \vee 3),\ \overline{4},\ (\overline{1} \vee 2 \vee 4),\ (\overline{1} \vee \overline{2} \vee \overline{3} \vee 4)$	T-learn
$\Longrightarrow$	FailState	fail

Lazyness in DPLL(T)

© Scott Adams, Inc./Dist. by UFS, Inc.

T-solver SAT solver

Outline

- Summary of Last Week
- Satisfiability Modulo Theories
- DPLL(T)
- Equality and Uninterpreted Functions in Practice

Example (SMT-LIB 2 for Propositional Logic)

formula $(x_1 \vee \neg x_3) \wedge (x_2 \vee x_3 \vee \neg x_1) \wedge (\neg x_1 \vee x_2 \vee x_3)$ can be expressed by

```
(declare-const x1 Bool)
(declare-const x2 Bool)
(declare-const x3 Bool)
(assert (or x1 (not x3)))
(assert (or x2 x3 (not x1)))
(assert (or (not x1) x2 x3))
(check-sat)
(get-model)
```

Propositional Logic in SMT-LIB 2

- `declare-const` *x* `Bool` creates propositional variable named *x*
- prefix notation for `and`, `or`, `not`, `implies`
- `assert` demands given formula to be satisfied
- `check-sat` issues satisfiability check of conjunction of assertions
- `get-model` prints model (after satisfiability check)

Example (SMT-LIB 2 for EUF)

$f(f(a)) \approx a \wedge f(a) \approx b \wedge \neg(a \approx b)$ is expressed as

```
(declare-sort A)
(declare-const a A)
(declare-const b A)
(declare-fun f (A) A)
(assert (= (f (f a)) a))
(assert (= (f a) b))
(assert (distinct a b))
(check-sat)
(get-model)
```

EUF in SMT-LIB 2

- terms must have sort, so declare fresh sort and use for all symbols: `declare-sort` S creates sort named S
- `declare-const` x s creates variable named x of sort S
- `declare-fun` F $(S_1 \dots S_n)$ T creates uninterpreted $F\colon S_1 \times \cdots \times S_n \to T$
- prefix notation as in `(f (f a))` to denote $f(f(a))$ and `(=` x y`)` for equality
- `(distinct` x y`)` is equivalent to `not(=` x y`)`

Example

$2x \geqslant y + z \land \neg(x \approx y)$ is expressed as

```
(declare-const x Int)
(declare-const y Int)
(declare-const z Int)
(assert (>= (* 2 x) (+ y z)))
(assert (not (= x y)))
(check-sat)
(get-model)
```

Integer Arithmetic in SMT-LIB 2

- `declare-const` x `Int` creates integer variable named x
- numbers `0`, `1`, `-1`, `42`,... are built-in
- `+`, `*`, `-` are $+_{\mathbb{Z}}$, $\cdot_{\mathbb{Z}}$, $-_{\mathbb{Z}}$, used in prefix notation: `(+ 2 3)`
- `=` also covers equality on $\mathbb{Z}$
- `<`, `<=`, `>`, `>=` are $<_{\mathbb{Z}}$, $\leqslant_{\mathbb{Z}}$, $>_{\mathbb{Z}}$, $\geqslant_{\mathbb{Z}}$

EUF in python/z3

```
A = DeclareSort('A') # new uninterpreted sort named 'A'
a = Const('a', A) # create constant of sort A
b = Const('b', A) # create another constant of sort A
f = Function('f', A, A) # create function of sort A -> A

s = Solver()
s.add(f(f(a)) == a, f(a) == b, a != b)

print s.check() # sat
m = s.model()
print "interpretation assigned to A:"
print m[A] # [A!val!0, A!val!1]
print "interpretations:"
print m[f] # [A!val!0 -> A!val!1, A!val!1 -> A!val!0, ...]
print m[a] # A!val!0
print m[b] # A!val!1
```

Example (Quantifiers and Monkeys)

In a village of monkeys every monkey owns at least two bananas: 

```
(declare-sort monkey)
(declare-sort banana)
(declare-fun owns (monkey banana) Bool)
(declare-fun b1 (monkey) banana)
(declare-fun b2 (monkey) banana)

(assert (forall ((M monkey)) (not (= (b1 M) (b2 M)))))
(assert (forall ((M monkey)) (owns M (b1 M))))
(assert (forall ((M monkey)) (owns M (b2 M))))
(assert (forall ((M1 monkey) (M2 monkey) (B banana))
  (implies (and (owns M1 B) (owns M2 B)) (= M1 M2))))
```

DPLL(T)

Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli.
Solving SAT and SAT Modulo Theories: From an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T).
Journal of the ACM 53(6), pp. 937–977, 2006.

Application

Miroslav N. Velev and Randal E. Bryant.
Bit-level abstraction in the verification of pipelined microprocessors by correspondence checking.
In Proc. of Formal Methods in Computer-Aided Design, pp. 18–35, 1998.