Model Checking, Winter Semester 2015/2016

Satisfiability Modulo Theories Overview

Version 2015.1

Armin Biere (biere@jku.at)
Martina Seidl (martina.seidl@jku.at)
Satisfiability Modulo Theories (SMT)

Example

\[ f(x) \neq f(y) \land x + u = 3 \land v + y = 3 \land u = a[z] \land v = a[w] \land z = w \]

- formulas in first-order logic
  - usually without quantifiers, variables implicitly existentially quantified
  - but with sorted / typed symbols and
  - functions / constants / predicates are interpreted
  - SMT quantifier reasoning weaker than in first-order theorem proving (FO)
  - much richer language compared to propositional logic (SAT)
- no need to axiomatize “theories” using axioms with quantifiers
  - important theories are “built-in”:
    - uninterpreted functions, equality, arithmetic, arrays, bit-vectors . . .
  - focus is on decidable theories, thus fully automatic procedures
- state-of-the-art SMT solvers essentially rely on SAT solvers
  - SAT solver enumerates solutions to a propositional skeleton
  - propositional and theory conflicts recorded as propositional clauses
  - DPLL(T), CDCL (T), read DPLL modulo theory T or CDCL modulo T
- SMT sweet spot between SAT and FO: many (industrial) applications
  - standardized language SMTLIB used in applications and competitions
int middle (int x, int y, int z) {
    int m = z;
    if (y < z) {
        if (x < y)
            m = y;
        else if (x < z)
            m = y;
    } else {
        if (x > y)
            m = y;
        else if (x > z)
            m = x;
    }
    return m;
}

dthis program is supposed to return the middle (median) of three numbers
Test Suite for Buggy Program

middle \( (1, 2, 3) = 2 \)
middle \( (1, 3, 2) = 2 \)
middle \( (2, 1, 3) = 1 \)
middle \( (2, 3, 1) = 2 \)
middle \( (3, 1, 2) = 2 \)
middle \( (3, 2, 1) = 2 \)
middle \( (1, 1, 1) = 1 \)
middle \( (1, 1, 2) = 1 \)
middle \( (1, 2, 1) = 1 \)
middle \( (2, 1, 1) = 1 \)
middle \( (1, 2, 2) = 2 \)
middle \( (2, 1, 2) = 2 \)
middle \( (2, 2, 1) = 2 \)

- This black box test suite has to be generated manually.

- How to ensure that it covers all cases?

- Need to check outcome of each run individually and determine correct result.

- Difficult for large programs.

- Better use specification and check it.
let $a$ be an array of size 3 indexed from 0 to 2

$$a[i] = x \land a[j] = y \land a[k] = z$$

$$\land$$

$$a[0] \leq a[1] \land a[1] \leq a[2]$$

$$\land$$

$$i \neq j \land i \neq k \land j \neq k$$

$$\rightarrow$$

$$m = a[1]$$

median obtained by sorting and taking middle element in the order coming up with this specification is a manual process
int m = z;
if (y < z) {
    if (x < y)
        m = y;
    else if (x < z)
        m = y;
} else {
    if (x > y)
        m = y;
    else if (x > z)
        m = x;
}
return m;

(y < z ∧ x < y → m = y)
∧
(y < z ∧ x ≥ y ∧ x < z → m = y)
∧
(y < z ∧ x ≥ y ∧ x ≥ z → m = z)
∧
(y ≥ z ∧ x > y → m = y)
∧
(y ≥ z ∧ x ≤ y ∧ x > z → m = x)
∧
(y ≥ z ∧ x ≤ y ∧ x ≤ z → m = z)
∧
(y ≥ z ∧ x ≤ y ∧ x ≤ z → m = z)
∧

this formula can be generated automatically by a compiler
Checking Specification as SMT Problem

<table>
<thead>
<tr>
<th>Let $P$ be the encoding of the program, and $S$ of the specification.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program is correct if $P \rightarrow S$ is valid.</td>
</tr>
<tr>
<td>Program has a bug if $P \rightarrow S$ is invalid.</td>
</tr>
<tr>
<td>Program has a bug if negation of $P \rightarrow S$ is satisfiable (has a model).</td>
</tr>
<tr>
<td>Program has a bug if $P \land \neg S$ is satisfiable (has a model).</td>
</tr>
</tbody>
</table>

\[
(y < z \land x < y \rightarrow m = y) \land \\
(y < z \land x \geq y \land x < z \rightarrow m = y) \land \\
(y < z \land x \geq y \land x \geq z \rightarrow m = z) \land \\
(y \geq z \land x > y \rightarrow m = y) \land \\
(y \geq z \land x \leq y \land x > z \rightarrow m = x) \land \\
(y \geq z \land x \leq y \land x \leq z \rightarrow m = z) \land \\
a[i] = x \land a[j] = y \land a[k] = z \land \\
a[0] \leq a[1] \land a[1] \leq a[2] \land \\
i \neq j \land i \neq k \land j \neq k \land \\
m \neq a[1]
\]
(set-logic QF_AUFLIA)
(declare-fun x () Int) (declare-fun y () Int) (declare-fun z () Int) (declare-fun m () Int)
(assert (=> (and (< y z) (< x y)) (= m y)))
(assert (=> (and (< y z) (>= x y) (< x z)) (= m y))) ; fix by replacing last 'y' by 'x'
(assert (=> (and (< y z) (>= x y) (>= x z)) (= m z)))
(assert (=> (and (>= y z) (> x y)) (= m y)))
(assert (=> (and (>= y z) (<= x y) (> x z)) (= m x)))
(assert (=> (and (>= y z) (<= x y) (<= x z)) (= m z)))
(declare-fun i () Int) (declare-fun j () Int) (declare-fun k () Int)
(declare-fun a () (Array Int Int))
(assert (and (<= 0 i) (<= i 2) (<= 0 j) (<= j 2) (<= 0 k) (<= k 2)))
(assert (and (= (select a i) x) (= (select a j) y) (= (select a k) z)))
(assert (<= (select a 0) (select a 1) (select a 2)))
(assert (distinct i j k))
(assert (distinct m (select a 1)))
(check-sat)
(get-model)
(exit)
Checking Middle Example with Z3

```
$ z3 middle-buggy.smt2
sat
(model
  (define-fun i () Int 1)
  (define-fun a () (Array Int Int) (_ as-array k!0))
  (define-fun j () Int 0)
  (define-fun k () Int 2)
  (define-fun m () Int 2281)
  (define-fun z () Int 2283)
  (define-fun y () Int 2281)
  (define-fun x () Int 2282)
  (define-fun k!0 ((x!1 Int)) Int
    (ite (= x!1 2) 2283
    (ite (= x!1 1) 2282
    (ite (= x!1 0) 2281 2283))))
)
```

```
$ z3 middle-fixed.smt2
unsat
see also http://rise4fun.com
```
Encoding with Bit-Vector Logic in SMTLIB2

(set-logic QF_AUFBV)
(declare-fun x () (_ BitVec 32)) (declare-fun y () (_ BitVec 32))
(declare-fun z () (_ BitVec 32)) (declare-fun m () (_ BitVec 32))
(assert (=> (and (bvult y z) (bvult x y) ) (= m y)))
(assert (=> (and (bvult y z) (bvuge x y) (bvult x z)) (= m y))) ; fix last 'y'->'x'
(assert (=> (and (bvult y z) (bvuge x y) (bvuge x z)) (= m z)))
(assert (=> (and (bvuge y z) (bvugt x y) ) (= m y)))
(assert (=> (and (bvuge y z) (bvule x y) (bvugt x z)) (= m x)))
(assert (=> (and (bvuge y z) (bvule x y) (bvule x z)) (= m z)))
(declare-fun i ()(_ BitVec 2)) (declare-fun j ()(_ BitVec 2)) (declare-fun k ()(_ BitVec 2))
(declare-fun a ()(Array (_ BitVec 2) (_ BitVec 32)))
(assert (and (bvule #b00 i) (bvule i #b10) (bvule #b00 j) (bvule j #b10)))
(assert (and (bvule #b00 k) (bvule k #b10)))
(assert (and (= (select a i) x) (= (select a j) y) (= (select a k) z)))
(assert (bvule (select a #b00) (select a #b01)))
(assert (bvule (select a #b01) (select a #b10)))
(assert (distinct i j k))
(assert (distinct m (select a #b01)))
(check-sat) (get-model) (exit)
Checking Middle Example with Boolector

$ boolector -m middle32-buggy.smt2
sat
x 1011100011111100101111101111110111
y 01111100011111100111110111110111
z 111110000111111011011111011111001
m 011111000111111001011111011111011
i 01
j 00
k 10
a[10] 11110000111111011011111011111001
a[01] 10111000111111001011111011111011
a[00] 011111000111111001011111011111011

$ boolector middle32-fixed.smt2
unsat

see also  http://fmv.jku.at/boolector
Theory of Uninterpreted Functions and Equality

- functions as in first-order (FO): sorted / typed without interpretation
- equality as single interpreted predicate
  - congruence axiom \( \forall x, y : x = y \rightarrow f(x) = f(y) \)
  - similar variants for functions with multiple arguments
  - always assumed in FO if equality is handled explicitly (interpreted)
- uninterpreted functions allow to abstract from concrete implementations
  - in hardware (HW) verification abstract complex circuits (e.g. multiplier)
  - in software (SW) verification abstract sub routine computation
- congruence closure algorithms using fast union-find data structures
  - start with all terms (and sub-terms) in different equivalence classes
  - if \( t_1 = t_2 \) is an asserted literal merge equivalence classes of \( t_1 \) and \( t_2 \)
  - for all elements of an equivalence class check congruence axiom
    - let \( t_1 \) and \( t_2 \) be two terms in the same equivalence class
    - if there are terms \( f(t_1) \) and \( f(t_2) \) merge their equivalence classes
  - continue until the partition of terms in equivalence classes stabilizes
  - if asserted disequality \( t_1 \neq t_2 \) exists with \( t_1, t_2 \) in the same equivalence class then unsatisfiable otherwise satisfiable
Example for Uninterpreted Functions and Equality

assume flattened structure where all sub-terms are identified by variables

\[ [x \mid y \mid t \mid u \mid v] \]

\[ x = y \land x = g(y) \land t = g(x) \land u = f(x, t) \land v = f(y, x) \land u \neq v \]

asserted literal \( x = y \) puts \( x \) and \( y \) in to the same equivalence class

\[ [x \mid t \mid u \mid v] \]

\[ x = y \land x = g(y) \land t = g(x) \land u = f(x, t) \land v = f(y, x) \land u \neq v \]

apply congruence axiom since \( x \) and \( y \) in same equivalence class

\[ [x \mid t \mid u \mid v] \]

\[ x = y \land x = g(y) \land t = g(x) \land \underbrace{u = f(x, t) \land v = f(y, x)}_{u \neq v} \land u \neq v \]

apply congruence axiom since \( y, x \) and \( t \) are all in same equivalence class

\[ [x \mid t \mid u \mid v] \]

\[ x = y \land x = g(y) \land t = g(x) \land u = f(x, t) \land v = f(y, x) \land u \neq v \]

\( u \) and \( v \) in the same equivalence class but \( u \neq v \) asserted

thus \textit{unsatisfiable}
Theory of Arrays

- functions “read” and “write”: \( \text{read}(a, i), \text{write}(a, i, v) \)
- axioms

\[
\forall a, i, j: i = j \rightarrow \text{read}(a, i) = \text{read}(a, j) \quad \text{array congruence}
\]

\[
\forall a, v, i, j: i = j \rightarrow \text{read}(\text{write}(a, i, v), j) = v \quad \text{read over write 1}
\]

\[
\forall a, v, i, j: i \neq j \rightarrow \text{read}(\text{write}(a, i, v), j) = \text{read}(a, j) \quad \text{read over write 2}
\]

- used to model memory (HW and SW)
- eagerly reduce arrays to uninterpreted functions by eliminating “write”

\[
\text{read}(\text{write}(a, i, v), j) \quad \text{replaced by} \quad (i = j \ ? \ v : \text{read}(a, j))
\]

- more sophisticated non-eager algorithms are usually faster
- such as for instance the lemmas-on-demand algorithm in Boolector
Simple Array Example

\[ i \neq j \land u = \text{read}(\text{write}(a, i, v), j) \land v = \text{read}(a, j) \land u \neq v \]

eliminate “write”

\[ i \neq j \land u = (i = j \ ? \ v : \text{read}(a, j)) \land v = \text{read}(a, j) \land u \neq v \]

simplify conditional by assuming “i \neq j”

\[ i \neq j \land u = \text{read}(a, j) \land v = \text{read}(a, j) \land u \neq v \]

applying congruence for both “read”

\[ i \neq j \land u = \text{read}(a, j) = \text{read}(a, j) = v \land u \neq v \]

which is clearly unsatisfiable
Theory of Bit-Vectors

- allows “bit-precise” reasoning
  - captures semantics of low-level languages like assembler, C, C++, ...
  - Java / C# also use two-complement representations for `int`
  - modelling of hardware / circuits on the word-level (RTL)
  - important for security applications and precise test case generation

- many operations
  - logical operations, bit-wise operations (and, or)
  - equalities, inequalities, disequalities
  - shift, concatenation, slicing
  - addition, multiplication, division, modulo, ...

- main approach is reduction to SAT through *bit-blasting*
  - reduction of bit-vector operations similar to circuit synthesis
  - Ackermann’s Reduction only needs equality and disequality
**Propositional Skeleton**

**Example (arbitrary LRA formula)**

\[
x \neq y \land (2 \times x \leq z \lor \neg (x - y \geq z \land z \leq y))
\]

eliminate \(\neq\) by disjunction

\[
\left(\underbrace{x < y}_a \lor \underbrace{x > y}_b\right) \land \left(\underbrace{2 \times x \leq z}_c \lor \neg \left(\underbrace{x - y \geq z}_d \land \underbrace{z \leq y}_e\right)\right)
\]

which is abstracted to a propositional formula called “propositional skeleton”

\[
(a \lor b) \land (c \lor \neg (d \land e))
\]

with \(\alpha(x < y) = a, \quad \alpha(x > y) = b, \ldots\)

SAT solver enumerates solutions, e.g., \(a = b = c = d = e = 1\)

check solution literals with theory solver, e.g., Fourier-Motzkin

spurious solutions (disproven by theory solver) added as “lemma”,

\(\neg(a \land b \land c \land c \land d \land e)\) or just \(\neg(a \land b)\) after minimization

continue until SAT solver says *unsatisfiable* or theory solver *satisfiable*
this is an extremely “lazy” version of DPLL (T) / CDCL(T)

\[ \text{LemmasOnDemand}(\phi) \]

\[ \psi = \text{PropositionalSkeleton}(\phi) \]

let \( \alpha \) be the abstraction function, mapping theory literals to prop. literals

while \( \psi \) has satisfiable assignment \( \sigma \)

let \( l_1, \ldots, l_n \) be all the theory literals with \( \sigma(\alpha(l_i)) = 1 \)

check conjunction \( L = l_1 \land \cdots \land l_n \) with theory solver

if theory solver returns satisfying assignment \( \rho \) return \text{satisfiable}

determine “small” sub-set \( \{k_1, \ldots, k_m\} \subseteq \{l_1, \ldots, l_n\} \) where

\[ K = k_1 \land \cdots \land k_m \] remains unsatisfiable (by theory solver)

add lemma \( \neg K \) to \( \psi \), actually replace \( \psi \) by \( \psi \land \alpha(\neg K) \)

return \text{unsatisfiable}

note that these lemmas \( \neg K \) are all clauses
SMT-Lib

SMT-Lib (www.smtlib.org) is a community portal for people working on and with SMT Solving including

- ... a standard for describing background theories and logics
  6 background theories, > 20 logics

- ... a standard for input/output of SMT solvers

- ... a collection of 95492 benchmark formulas
totalling 59.2 GB in 383 families over 22 logics

- ... a collection of tools

- ... the basis of the annual competition
SMT Solvers

- aim of an SMT solver: check satisfiability of formula $\phi$
  - not over all (first-order) interpretations
  - but with respect to some background theory

- artifacts of an SMT solving system compliant to SMTLib v2:
  - based on many-sorted first-order logic with equality
  - background theory: taken from catalogue of theories
    - basic theories
    - combined theories
  - interface: command language
  - input formula
The SMT-Lib Command Language

- communication with the SMT solver
  - textual input channel
  - two textual output channels
    - regular output
    - diagnostic output

- primary design goal: interaction between programs

- types of commands
  - defining sorts and functions
  - managing assertions
  - checking satisfiability
  - setting options
  - getting information
  - exit

- responses: unsupported, success, error \langle string \rangle
Theories and Logics

A theory

- ... defines a vocabulary for sorts and functions (signature).
- ... associates each sort with literals.
- ... may be infinite.
- ... has often an informal specification (in natural language).

A logic

- ... consists of at least one theory.
- ... restricts the kind of expressions to be used.
- ... has often an informal specification (in natural language).

SMTLib provides various theories and logics.
Some Logics without Quantifiers

<table>
<thead>
<tr>
<th>Logic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>QF_UF</td>
<td>formulas over uninterpreted functions</td>
</tr>
<tr>
<td>QF_LIA</td>
<td>formulas over linear integer arithmetic</td>
</tr>
<tr>
<td>QF_NIA</td>
<td>formulas over integer arithmetic</td>
</tr>
<tr>
<td>QF_BV</td>
<td>formulas over fixed-size bitvectors</td>
</tr>
<tr>
<td>QF_ABV</td>
<td>formulas over bitvectors and bitvector arrays</td>
</tr>
<tr>
<td>QF_AUFBV</td>
<td>formulas over bitvectors and bitvector arrays with unint. func.</td>
</tr>
<tr>
<td>QF_AUFLIA</td>
<td>linear formulas over integer arrays with uninterpreted functions</td>
</tr>
</tbody>
</table>
Terms, Functions, and Predicates

- Structure of terms and functions:
  - \( \langle \text{constant} \rangle \)
  - \( \langle \text{identifier} \rangle \)
  - as (\( \langle \text{identifier} \rangle \langle \text{sort} \rangle \))
  - (\( \langle \text{identifier} \rangle \langle \text{term} \rangle^+ \))
  - (as (\( \langle \text{identifier} \rangle \langle \text{sort} \rangle \)) \( \langle \text{term} \rangle^+ \))
  - quantifier terms with forall, exists
  - attributed terms!
  - bound terms with let

- example \((\text{or} (\langle \text{p} \rangle (\langle + \rangle \langle \text{q} \rangle \langle 2 \rangle \rangle )) (\langle < \rangle \langle \text{p} \rangle (\langle - \rangle \langle \text{q} \rangle \langle 2 \rangle \rangle ))\))

- terms are always typed
- no syntactic difference between functions and predicates
Declaring Functions (and Constants)

- \texttt{declare-fun} \ (\sigma_1 \ldots \sigma_n) \ \sigma: 
  - declaration of new function with \( n \) parameters of sorts \( \sigma_1 \ldots \sigma_n \)
  - return value of sort \( \sigma \)
- constants are 0-ary functions

Example

- \texttt{(declare-fun x () Bool)}
- \texttt{(declare-fun f (Int Int) Bool)}
- \texttt{(declare-fun ff ( (Int Int Bool) ) Int)}
Satisfiability Commands

- **(assert \(\text{term}\))**
  - term is of sort Bool
  - solver shall assume that term is true

- **(check-sat)**
  - check consistency of conjunction of assertions
  - response: sat, unsat, unknown

- get a solution with **(get-model)**

**Example**

(set-option :model true)
(declare-fun x () Int)
(assert (>= (* 3 x) (+ x x)))
(check-sat)
(get-model)
Example: Boolean Expressions

- **Boolean expressions** are defined in the *Core Theory*
- **sort:** `Bool`
- **constants:** `true, false` *(both of sort `Bool`)*
- **functions:**
  - `not`
  - `or, xor, and, =>`
  - `=, distinct` *(equality, inequality)*
  - `ite` *(if-then-else)*

**Example**

```
(set-logic QF_UF)
(declare-fun x () Bool)
(declare-fun y () Bool)
(assert (and (or x (not y)) (or (not x) y)))
(check-sat)
(exit)
```
Example: Real Expressions

- Real expressions are defined in the *Real Theory*
- **sort:** Real
- **constants:** numerals, decimals (all of sort Real)
- **functions with signature:**
  - (- (Real) Real) ; negation
  - (- (Real Real) Real ) ; subtraction
  - (+ (Real Real) Real )
  - (*) (Real Real) Real )
  - (/ (Real Real) Real )
  - (<= (Real Real) Bool)
  - (< (Real Real) Bool)
  - (>= (Real Real) Bool)
  - (> (Real Real) Bool)

Example

(set-logic QF_LRA)
(declare-fun x () Real)
(declare-fun y () Real)
(assert (and (>= (* 2 x) (+ y 3.2)) (= x y)))
(check-sat)
Example: Array Expressions

The theory of Arrays defines functions to read and write elements of arrays.

- sort: Array <sort of index> <sort of elements>
- functions
  - (select (array index) value) where
    - array is of sort (Array <sort of index> <sort of elements>)
    - index is of sort <sort of index>
    - value is of sort <sort of elements>
  - (store (array1 index value) array2) where
    - array1, array2 are of sort (Array <sort of index> <sort of elements>)
    - index is of sort <sort of index>
    - value is of sort <sort of elements>

Example

(declare-fun a () (Array Int Bool))
(declare-fun b () (Array Int Bool))
(assert (= (select a 1 ) true))
(assert (= (store b 1 false) a))
(check-sat) ; result is unsat
Example: Fixed-Sized Bitvectors Expressions

- **sort:** \((\_ \text{BitVec } n)\) where \(n\) is the size of the bitvector
- **functions:**
  - \((\text{op1 } (\_ \text{BitVec } m) (\_ \text{BitVec } m))\)
    - with \(\text{op1} \in \{\text{bvnot, bvneg}\}\)
  - \((\text{op2 } (\_ \text{BitVec } m) (\_ \text{BitVec } m) (\_ \text{BitVec } m))\)
    - with \(\text{op2} \in \{\text{bvand, bvor, bvadd, bvmul, bvdiv, bvurem, bvshl, bvlshr}\}\)
  - \((\text{bvult } (\_ \text{BitVec } m) (\_ \text{BitVec } m) \text{ Bool})\)
    - binary comparison
  - \((\_ \text{extract } i \ j) (\_ \text{BitVec } m) (\_ \text{BitVec } n))\)
    - extract contiguous subvector from index \(i\) to index \(j\)
  - \((\text{concat } (\_ \text{BitVec } i) (\_ \text{BitVec } j) (\_ \text{BitVec } m))\)
    - combines two bitvectors
Outlook

SMTLib2 offers many more language concepts, for example:

- Makros
- User-defined sorts
- Many Options
- Scopes
- ...

More infos:

The SMT4J Solver

**Approach:** Lemmas on Demand

- implemented in Java
- SMTLib v2 compliant
- modular (support different theories)
- performant, maintainable, simple
- lazy (later maybe mixed)
- easy to integrate in application programs