Formal Methods in Computer Science
UE 185.A93 WS 2017/18
Block 2: Satisfiability

Florian Lonsing

Knowledge-Based Systems Group, Vienna University of Technology, Austria
http://www.kr.tuwien.ac.at/staff/lonsing/

This work is supported by the Austrian Science Fund (FWF) under grant S11409-N23.

Satisfiability Modulo Theories (SMT)

- Satisfiability checking of a first-order logic (FOL) formula with equality with respect to a background theory.
- E.g., theory $\mathcal{T}_{cons}$ of lists: theory of equality $\mathcal{T}_E$ + list-specific axioms.
- Theories of interest: integers, real numbers, arrays, bitvectors,...
- Most theories are quantifier-free.
- Potentially higher expressiveness compared to propositional logic.
- Specialized theory-specific inference methods.
- Practically relevant in industry: verification, model checking,...

Example

The formula $(x - y) > 0 \leftrightarrow (x > y)$ is valid over the theory of integers but not valid over the theory of fixed-size bitvectors, i.e., modular arithmetic with under/overflow.
Counterexample: $x := \#000$, $y := \#110$, $x - y = \#010$.

SMT-LIB: Satisfiability Modulo Theories Library

Website: `http://smtlib.cs.uiowa.edu/index.shtml`

> *SMT-LIB is an international initiative aimed at facilitating research and development in Satisfiability Modulo Theories.*

Goals:

- Provide standard rigorous descriptions of background theories.
- Promote common input and output languages for SMT solvers.
- Develop a community of researchers and users of SMT technology.
- Make available a large library of benchmarks for SMT solvers.
- Collect and promote software tools useful to the SMT community.

Z3: SMT Solver and Theorem Prover

Website: `https://github.com/Z3Prover/z3`

- SMT solver and theorem prover developed by Microsoft Research.
- Supports input in the SMT-LIB format (and variations thereof).
- API to construct formulas over various theories and logics.

⇒ In our exercises of block 2 (satisfiability) we want to apply Z3 to model and solve problems related to the lecture.

Recommended Z3 Tutorial: `https://rise4fun.com/z3/tutorial`

Further Resources:

- `http://smtlib.github.io/jSMTLIB/SMTLIBTutorial.pdf`
- `http://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.6-r2017-07-18.pdf`

Z3: Input Language and Basic Use (1/2)

- LISP-like syntax: *s-expressions*.
- E.g., $a \times (b + c)$ is represented as $(\times\ a\ (+\ b\ c))$.
- Prefix notation: function (e.g., $\times$, $+$) followed by arguments.
- Basic building blocks of SMT formulas: constants and functions.
- Constants are just functions that take no arguments.

Z3: Input Language and Basic Use (1/2)

Z3 input:

```
; lines like these prefixed with ';' are comments
; define integer constants 'a', 'b', 'c', and 'res'
(declare-const a Int)
(declare-const b Int)
(declare-const c Int)
(declare-const res Int)
; compute 'res := a * (b + c)'
(assert (= res (* a (+ b c))))
(check-sat)
; extract a model of the formula
(get-model)
```

Z3: Input Language and Basic Use (1/2)

Z3 output: model (i.e., concrete interpretation)

```
sat
(model
  (define-fun a () Int
    0)
  (define-fun res () Int
    0)
  (define-fun c () Int
    0)
  (define-fun b () Int
    0)
)
```

Z3: Input Language and Basic Use (2/2)

- Z3 maintains a stack of assertions added by `assert` function.
- Each call of `assert` *conjunctively* adds a new assertion to the stack.
- `check-sat`: check if there is a model wrt. *all* added assertions.
- `check-sat` checks for satisfiability, not validity.
- Validity checking: check satisfiability of the negation of a formula.
- E.g., to prove validity of $(\alpha \wedge \beta) \rightarrow \gamma$, first assert α, then β, and finally $\neg\gamma$, and call `check-sat`.

Propositional Logic

- Basic Boolean operators.
- Constants `true` and `false`.
- Negation `not`.
- Implication (`=>`), disjunction (`or`), conjunction (`and`), and exclusive OR (`xor`).
- Definition of Boolean functions by `define-fun` allows to build arbitrarily complex subformulas (see example on next slide).

Propositional Logic

We prove that the formula $(x - y) > 0 \leftrightarrow (x > y)$ from the first slide is valid over the theory of integers.

Z3 input:

```
(declare-const x Int)
(declare-const y Int)
(declare-const s Int)
(define-fun LHS () Bool
  (> (- x y) 0) )
(define-fun RHS () Bool
  (> x y) )
(define-fun IFF () Bool
  (and
    (=> LHS RHS)
    (=> RHS LHS)
  ) )
(assert (not IFF))
(check-sat)
```

Fixed Size Bitvectors

- Ordered sequence $\langle b_{n-1}, b_{n-2}, \ldots, b_0 \rangle$ of n bits.
- Like binary numbers in computers: applications in verification.
- Bitvector constants:
  - Must be declared with constant size (i.e., number of bits).
  - Binary or hexadecimal notation: `#b010`, `#xf`.
- Bitvector operations: sizes of operands must match.
- Addition (`bvadd`), subtraction (`bvsub`), multiplication (`bvmul`).
- Division (`bvudiv`, `bvsdiv`) and remainder (`bvurem`, `bvsrem`).
- Relational operators: $<$ (`bvult`), $\leq$ (`bvule`), $>$ (`bvugt`), $\geq$ (`bvuge`), and respective signed variants, e.g.:
  - `(bvslt #b111 #b000)` is true.
  - `(bvult #b111 #b000)` is false.

Fixed Size Bitvectors

Z3 input:

```
; We want to prove that bitvector division by two
; is equivalent to bitvector logical right shift by 1
(declare-const x (_ BitVec 8))
(declare-const res_shift (_ BitVec 8))
(declare-const res_div (_ BitVec 8))
; compute res_div := x / 2
; the size of the constant '2' must be equal to
; the size of 'x'
(assert (= (bvudiv x #b00000010) res_div))
; compute res_shift := x >> 1
(assert (= (bvlshr x #b00000001) res_shift))
; check if 'res_shift == res_div' is valid
(assert (not (= res_shift res_div)))
(check-sat)
```

Z3 output: unsat

Integers

- Signed integers.
- Arithmetic operators: addition (+), subtraction (-), multiplication (*), division (`div`), modulo (`mod`).
- Relational operators: <, >, <=, >=.
- Usual semantics of arithmetic operations (see example on next slide).

Integers

Z3 input:

```
(declare-const x Int)
(declare-const a Int)
(declare-const b Int)
(declare-const res1 Int)
(declare-const res2 Int)
; compute res1 := x * (a + b)
(assert (= (* x (+ a b)) res1))
; compute res2 := x * a + x * b
(assert (= (+ (* x a) (* x b)) res2))
; check validity of res1 = res2, which is the
; case due to the built-in distributivity axioms
; of * and + in theory of integers
(assert (not (= res1 res2)))
(check-sat)
```

Z3 output: unsat

Equality and Uninterpreted Functions

- Similar to the theory $\mathcal{T}_E$ presented in the lecture.
- Equality predicate (=): function that takes two arguments of the same sort (e.g., integers, bitvectors,...) and returns a Boolean value.
- Arbitrary sorts can be defined.
- Uninterpreted functions can be defined (`declare-fun`).
- Usual semantics ($\mathcal{T}_E$ axioms), see examples on next slide.

Equality and Uninterpreted Functions

Z3 input:

```
; declare a new custom sort 'MySort'
(declare-sort MySort)
; declare constants of type 'MySort'
(declare-const x MySort)
(declare-const y MySort)
; declare an uninterpreted function 'F' that maps a
; value of type 'MySort' to a value of type 'MySort'
(declare-fun F (MySort) MySort)
; check validity of 'x == y => F(x) == F(y)', which
; holds due to functional consistency.
(assert (= x y))
(assert (not (= (F x) (F y))))
(check-sat)
```

Z3 output: unsat

Equality and Uninterpreted Functions

*(Compare to previous example on distributivity of * over + in theory of integers.)*

Z3 input (formula is satisfiable):

```
(declare-const x Int)
(declare-const a Int)
(declare-const b Int)
(declare-const res1 Int)
(declare-const res2 Int)
; declare a new uninterpreted integer operator
; (i.e., a function) 'myop' with two arguments
(declare-fun myop (Int Int) Int)
; compute res1 := myop (x, (a + b))
(assert (= (myop x (+ a b)) res1))
; compute res2 := myop(x, a) + myop (x, b)
(assert (= (+ (myop x a) (myop x b)) res2))
; check whether 'res1 == res2', which is NOT the
; case since the theory of integers has no axioms
; for our custom operator 'myop'
(assert (not (= res1 res2)))
(check-sat)
```

Exercise Sheet

Schedule in WS 2017/18:

- November 7: presentation of exercise sheet and introduction.
- November 27: submission deadline (upload in TUWEL).
- December 12: presentation of solutions and feedback.

Important Notes:

- Please keep in mind the general information and guidelines presented during the kick-off meeting (slides available in TUWEL).
- Please follow the submission instructions and guidelines on the exercise sheet (available in TUWEL).