SMT Solvers

A CTF-oriented introduction

Giovanni Lagorio

giovanni.lagorio@unige.it
zxgio@zenhack.team
https://zxgio.sarahah.com

DIBRIS - Dipartimento di Informatica, Bioingegneria, Robotica e Ingegneria dei Sistemi
University of Genova, Italy

May 11, 2018

www.zenhack.team

Outline

Boolean SATisfiability problem

SAT – boolean satisfiability problem

The problem of determining whether there exists an interpretation that satisfies a given boolean formula; a formula that can be evaluated to true is said to be satisfiable

Eg. `(a and not b)` is satisfiable, `(a and not a)` is not

https://en.wikipedia.org/wiki/Boolean_satisfiability_problem

SAT

- is *NP-complete*: all NP problems are at most difficult to solve as SAT
- no known algorithm can *efficiently* solves each SAT problem
- *however*, heuristical SAT-algorithms can solve many problems involving thousands of variables and formulas with millions of symbols

Satisfiability Modulo Theories

SMT – Satisfiability Modulo Theories

Decision problem for logical formulas wrt combinations of theories, expressed in classical first-order logic with equality. Examples of theories typically are real numbers, integers, bit-vectors, ...

Eg. `(x + y < 12)` is satisfiable, `((x <= z) and (x > x))` is not

https://en.wikipedia.org/wiki/Satisfiability_modulo_theories

SMT

- can be *undecidable* (depends on the theories)
- SMT solvers
  - extends heuristic SAT solvers
  - can solve many real world problems
  - can be used to analyze and verify software

Outline

Z3

Z3

- open-source solver from Microsoft Research
- extremely efficient
- bindings for many languages, including Python

Installation Ubuntu-derived OS

- `sudo apt install python-virtualenv`
- `python2 -mvirtualenv z3 && . z3/bin/activate`
- optional: `pip install ipython`
- `git clone https://github.com/Z3Prover/z3.git && cd z3`
- `python scripts/mk_make.py -python && cd build && make install`

. . . it will take a while, enjoy a coffee ☺

Basic usage (1/2)

Let's get started:

- import symbols: `from z3 import *`
- create a solver: `s = Solver()`
  - collects constraints into a huge formula
  - checks satisfiability and find a model
- create variables: `x, y = Ints('x y')`
  - `x` is a (Python) object representing a Z3 integer variable named x
  - `y` is a (Python) object representing a Z3 integer variable named y
  - using *operator overloading* a (Python) expression like `x + y` gets evaluated into an object representing the expression $(x + y)$ in Z3
- add constraints: `s.add(x + y < 12)`

Basic usage (2/2)

- check satisfiability: `s.check()`
  `check` can either return:
  - `z3.sat`
  - `z3.unsat`
  - `z3.unknown`

  or work for a *long long* time
- (if sat) see the assignment: `s.model()`
  - the model is a dictionary from variables (named-constants) to values

Use `solve` to simply print out a solution

```
x, y = Ints('x y')
solve(x + y < 12)
```

Sort, types and values – Booleans

Z3 offers *a lot* of types, we discuss the main ones only

- Beware of terminology: every expression has a sort (= Z3 type)

  ```
  x, y = Ints('x y')
  x.sort()          # Int
  (x+y).sort()      # Int
  (x+y==3).sort()   # Bool
  ```

  and, obviously, a different Python type (`z3.ArithRef` and `z3.BoolRef`)
- you can create (Python objects that represent) values and variables
- `BoolSort()`, AKA booleans

  ```
  t, f, bv = BoolVal(True), BoolVal(False), Bool('bv')
  # is_bool can query the sort, is_true and is_false the value
  assert is_bool(t) and is_true(t) and not is_false(t)
  assert is_bool(f) and not is_true(f) and is_false(f)
  assert is_bool(bv) and not is_true(bv) and not is_false(bv)
  assert assert bool(t) and not bool(f)
  ```

Sort, types and values – Integers

- `IntSort()`, AKA integers AKA $\mathbb{Z}$

```
i, iv = IntVal(42), Int('iv')
assert is_int(i) and is_int_value(i) and i.as_long() == 42
assert is_int(iv) and not is_int_value(iv)
```

  - Inside Z3 convert to/from `RealSort()` with functions `ToReal`/`ToInt`

- `BitVecSort(`n`)`, AKA n-bit vectors AKA $\mathbb{Z}_{2^n}$

```
v, vv = BitVecVal(5, 3), BitVec('vv', 3)
assert is_bv(v) and is_bv_value(v) and \
       v.as_long() == 5 and v.as_signed_long() == -3
assert is_bv(vv) and not is_bv_value(vv)
```

  - bit vectors support bit-wise logical/shift operations (integers don't)
  - (most?) operations work with vectors of the same length
    - `Extract(`h`, `l`, `v`)` extracts bits $(l, l+1, \ldots, h)$ of v;
      e.g. `Extract(3, 0, bv)` returns the 4 least significant bits of `bv`
    - `ZeroExt(`n`, `v`)` adds n zero-bits
    - `Concat(`v_1`,`v_2`,...,`v_n`)` concatenates $v_1, \ldots, v_n$
  - there is a `StringSort`, but "strings" usually encoded as bit-vectors

Sort, types and values – Arrays

- Arrays in Z3 are used to model unbounded or very large arrays
  - For small collections it is more efficient to create different variables; e.g. `IntVector('x', 3)` returns a list of three `Int` variables named `x__0`, `x__1` and `x__2`
- `Array(`*name*`,` *dom*`,` *rng*`)` creates an array constant named *name* with given domain and range sorts; e.g.
  `a = Array('a', IntSort(), IntSort())`
- `Select(`*a*`,` *i*`)` returns the value stored at position *i* of the array *a*
  - We can also write `Select(a, i)` as `a[i]`
- `Store(`*a*`,` *i*`,` *v*`)` returns a new array identical to *a*, but on position *i*, where it contains the value *v*

Sort, types and values – Reals

- `RealSort()` — AKA Z3 reals
  - `RealVal` creates a Z3 real value from an int, long, float or string representing a number in decimal or rational notation, and
  - `Q` creates a Z3 rational from numerator and denominator
- To extract a Python value from a Z3 real x, you need to distinguish whether x is rational or algebraic

```
s = Solver()
x, y = Reals('x y')
s.add(2*x == 1, y*y == 2, y >= 0)
s.check()  # sat
m = s.model()  # m is [y = 1.4142135623?, x = 1/2]
x_sol, y_sol = m[x], m[y]
assert is_rational_value(x_sol)
assert is_algebraic_value(y_sol)
x_sol.numerator_as_long()   # 1
x_sol.denominator_as_long() # 2
y_rat_sol = y_sol.approx(10)
y_rat_sol.numerator_as_long()   # 388736063997
y_rat_sol.denominator_as_long() # 274877906944
388736063997.0/274877906944     # 1.4142135623733338
```

Pitfalls in building expressions

We already saw:

- Most often, overloading is used to build constraints; e.g. `y*y==2` builds a `z3.BoolRef` object corresponding to $y \cdot y = 2$

Pitfalls

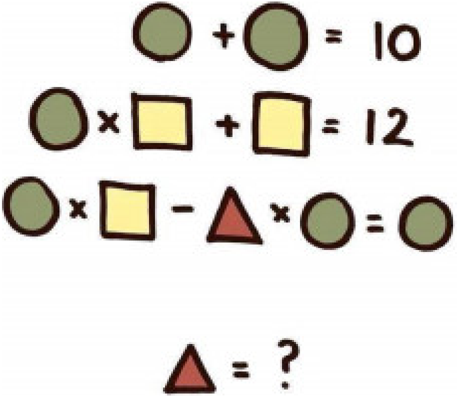
- boolean operators (`and`, `or` and `not`) *cannot* be overloaded; use `And`, `Or` and `Not` to build boolean expressions
  - there is also a `Implies` shortcut
- >> is an *arithmetic* shift; use `LShR` for the logical one
- /, %, <, <=, > and >= on bit-vectors are *signed*; use, respectively, `UDiv`, `URem`, `ULT`, `ULE`, `UGT` and `UGE` for unsigned ones

Special cases

- `Distinct(*args)`, for expressing the fact that all args are different
- `If(cond, then-exp, else-exp)` creates an if-then-else expression; eg. `def abs_z3(a): return If(a >= 0, a, -a)`

Exercise: geometric figures

Circle, square and triangle are integers



Taken from: `https://yurichev.com/writings/SAT_SMT_draft-EN.pdf`
(a suggested read, BTW)

Example: optimization

Can we prove the following claim?

`(x & (x - 1)) == 0` iff "`x` is a power of two"

```
x = BitVec('x', 32)
quick_check = (x & (x - 1)) == 0
x_is_power = Or([x==p for p in [2**i for i in range(32)]])
s = Solver()
s.add( quick_check == x_is_power )  # ok?
# nope! We should check whethere there is any counter-example:
s = Solver()
s.add( quick_check != x_is_power )
s.check()  # sat; yep, these are NOT equivalent
# shortcut:
prove( quick_check == x_is_power )
prove( And(x != 0, quick_check) == x_is_power )
```

(taken from `https://ericpony.github.io/z3py-tutorial/guide-examples.htm`)

Exercise: optimization

```
x = BitVec('x', 32)
y = BitVec('y', 32)

# Claim: (x ^ y) < 0 iff x and y have opposite signs
```

Can you (dis)prove this claim?

(taken from `https://ericpony.github.io/z3py-tutorial/guide-examples.htm`)

Save and restore the state

```
s = Solver()
x, y = Ints("x y")
s.add(x + y < 12, x >= 0)

s.push()       # creates a backtracking point
s.add( x==3 )
s.check()      # sat
s.model()[y]   # 0
s.pop()        # backtrack to the previously saved point

s.push()
s.add( x==15 )
s.check()      # sat
s.model()[y]   # -4
s.pop()

s.add( y==13 )
s.check() # unsat
```

Enumerating all solutions

```
s = Solver()
x, y = Ints("x y")
s.add(x + y < 12, x >= 0, x<= 8, y>=3, y<=10)
while s.check() == sat:
    m = s.model()
    print(m)
    s.add(Or(x != m[x], y != m[y]))
```

prints:

```
[y = 3, x = 0]
[y = 4, x = 1]
[y = 3, x = 1]
[y = 3, x = 2]
[y = 3, x = 3]
...
```

See also `https://stackoverflow.com/a/11869410`

Other features

You can also

- `simplify` an expression (lots of options: see `help_simplify()`); e.g.
  `simplify(3*x + 5 * y == 2*x - 7 + y) # x + 4*y == -7`
- `(Optimize.)minimize` an objective function
- ...

See `https://github.com/ericpony/z3py-tutorial`

Exercise: system of equations

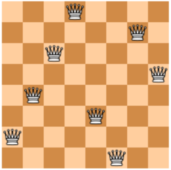
Solve the following system:

$$
\begin{aligned}
x_2 + 2\,x_3 - x_4 &= 1 \\
x_1 + x_3 + x_4 &= 4 \\
-x_1 + x_2 - x_4 &= 2 \\
7\,x_2 + 3\,x_3 - x_4 &= 7
\end{aligned}
$$

How many solutions can Z3 find? What happens if you remove an equation?

Example: The Eight Queens puzzle

> *. . . the problem of placing eight queens on an 8x8 chessboard so that no two queens attack each other. Thus, a solution requires that no two queens share the same row, column, or diagonal.*



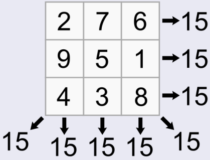
https://ericpony.github.io/z3py-tutorial/guide-examples.htm

```python
# We know each queen must be in a different row.
# So, we represent each queen by a single integer: the column position
Q = [ Int('Q_%i' % (i + 1)) for i in range(8) ]
val_c = [ And(1 <= Q[i], Q[i] <= 8) for i in range(8) ] # cols in interval [1, 8]
col_c = [ Distinct(Q) ] # at most one queen per column
# Diagonal constraint
diag_c = [ And(Q[i] - Q[j] != i - j, Q[i] - Q[j] != j - i)
           for i in range(8) for j in range(i) ]
solve(val_c + col_c + diag_c)
```

Exercise: magic squares

Magic squares

A magic square is a $n \times n$ grid filled with distinct positive integers in the range $1 \dots n^2$, s.t. each cell contains a different integer and the sum of each row, column and diagonal is equal to a "*magic constant*".
Magic squares are also called normal magic squares, in the sense that there are non-normal ones where integers are not restricted. E.g. (normal):



https://en.wikipedia.org/wiki/Magic_square

Write a program that produces a 3x3 (non-normal?) magic square for a given constant (if it exists). Generalize your program to $n \times n$ squares.

Outline

A simple encoding

Some programs can be easily "translated" into a formula; for instance we could encode `f` into a formula F

```
int f(int x, int y)
{
        int k = x + 1;
        int q = y - x;
        return k + 2 * q;
}
```

```
s = Solver()
x, y, k, q, result = \
 BitVecs('x y k q result', 32)
s.add( k == x + 1 )
s.add( q == y - x )
s.add( result == k + 2 * q )
```

then, we can use F an interpreter; e.g.

```
s.push()
s.add(x == 1, y == 2)
s.check()
print(s.model()[result]) # gives us f(1, 2) == 4
s.pop()
```

more interestingly. . .

We can “invert” the function

```
def invert_f(o): # returns x and y, s.t. f(x, y)==o
        s.push()
        s.add(result == o)
        if s.check() == sat:
                m = s.model()
                r = m[x], m[y]
        else:
                r = None
        s.pop()
        return r

invert_f(4)   # (1, 2)
invert_f(123) # (18, 70)
invert_f(42)  # (3, 22)
```

SSA Form

Can the previous translation work with this `f`?

```
int f(int x, int y)
{
        x = x + 1;
        y = y - x;
        return x + 2 * y;
}
```

```
int f(int x_0, int y_0)
{
        x_1 = x_0 + 1;
        y_1 = y_0 - x_1;
        return x_1 + 2 * y_1;
}
```

SSA: Static Single Assignment

> *...each variable is assigned exactly once, ... Existing variables in the original IR are split into versions ...*

https://en.wikipedia.org/wiki/Static_single_assignment_form

Branches

```
int f(int x, int y)
{
   x = x + 1;
   if x > y:
      y = y - x;
   else
      x *= 3;
   return x + 2 * y;
}
```

```
int f(int x_0, int y_0)
{
   x_1 = x_0 + 1;
   if x_1 > y_0:
      y_1 = y_0 - x_1;
   else
      x_2 = x_1 * 3;
   return x_??? + 2 * y_???;
}
```

Φ-functions

```
int f(int x, int y)
{
   x = x + 1;
   if x > y:
      y = y - x;
   else
      x *= 3;
   return x + 2 * y;
}
```

```
int f(int x_0, int y_0)
{
   x_1 = x_0 + 1;
   if x_1 > y_0:
      y_1 = y_0 - x_1;
   else
      x_2 = x_1 * 3;
   x_3 = Φ(x_1, x_2);
   y_2 = Φ(y_0, y_1);
   return x_3 + 2 * y_2;
}
```

Not real functions, can be encoded in Z3 by using `If`:

```
x0, x1, x2, x3, y0, y1, y2, result = BitVecs('x0 x1 x2 x3 y0 y1 y2 result', 32)
s = Solver()
s.add( x1 == x0 + 1 )
s.add( y1 == y0 - x1 )
s.add( x2 == x1 * 3)
s.add( x3 == If(x1 > y0, x1, x2) )
s.add( y2 == If(x1 > y0, y1, y0) )
s.add( result == x3 + 2 * y2 )
```

Generating fresh names

```python
class VarGenerator(object):
    def __init__(self):
        self.z3vars = {}
    def create(self, z3class, name, *z3args):
        v = z3class(name + "_0", *z3args)
        self.z3vars[v] = 0, name, z3class, z3args
        return v
    def fresh(self, v):
        i, name, z3class, z3args = self.z3vars[v]
        del self.z3vars[v]
        i += 1
        v = z3class(name + "_" + str(i), *z3args)
        self.z3vars[v] = i, name, z3class, z3args
        return v
# ...
v = VarGenerator()
a = v.create(Int, 'a')
b = v.create(BitVec, 'b', 32)
for _ in range(3):
    a, b = v.fresh(a), v.fresh(b)
    print(a, b)
```

Limitations

In general, imperative programs are not directly representable as first-order logic formulas because of unbounded

- loops
- recursion

(bounded loop/function-calls can be "inlined")

Exercise: find a and b

```python
def g(a, b):
    for i in range(10):
        if a > b:
            a -= b // 2
            b *= -1
        else:
            b += a + 10
    return a+b
```

Can you find `a` and `b` s.t. `g(a, b) == 0`?

Exercise: find the winning s

```
from pwn import u32
def f(s):
    win = 0
    j = 1
    k = 2
    if len(s) == 8:
        w1 = u32(s[:4], endian='big')
        w2 = u32(s[4:], endian='big')
        if w1 ^ w2 == 0x3b060569:
            k = w1 % 0xb4cadc9
        if w2 > w1:
            j = w2 % 0x2753f
    if j + k == 0:
        win = 1
    return win
```

Hint: try to understand the code and directly model its behaviour, without converting to SSA