CSE4051: Program Verification

Applications of SMT

2025 Fall

Woosuk Lee

Review: Satisfiability Modulo Theory (SMT)

- A first-order theory T is defined by the two components:
  - **Signature**: a set of nonlogical symbols. Given a signature Σ, a Σ-formula is one whose nonlogical symbols are from Σ. Signature restricts the syntax.
  - **Axioms**: A set of closed FOL formulas whose nonlogical symbols are from Σ. Axioms restrict the interpretations.
- Given a theory T with signature Σ and axioms A, an interpretation I is called T-interpretation if
  - $I \models a$ for every $a \in A$ (every axiom in A is valid under I)
- A Σ-formula F is T-satisfiable (or satisfiable modulo T) if there exists a T-interpretation that satisfies F.
- An SMT problem is to find a T-interpretation for a given formula in theory T.

Exercises

- In this lecture, we will use Z3 as done in the previous lecture.

Review: Using Z3Py

- Install Z3Py using pip: `pip install z3-solver`
- Import Z3Py in your Python script:

```
from z3 import *
```

- Define Boolean variables:

```
a = Bool("a")
b = Bool("b")
```

- Create logical formulas using Z3Py:

```
f1 = And(Not(a), Not(b))
f2 = Or(a, b)
```

- Solve the satisfiability problem:

```
solve(Not(f1 == f2))
```

- The `solve` function will return whether the formula is satisfiable or not, and if it is, it will provide an interpretation that satisfies the formula.

Arithmetic

```
from z3 import *
x=Int('x')
y=Int('y')
solve (x > 2, y < 10, x + 2*y == 7)
x = Real('x')
y = Real('y')
solve(x**2 + y**2 > 3, x**3 + y < 5)
```

Bitvectors

```
x = BitVec('x', 32)
y = BitVec('y', 32)
solve(x & y == ~y)
solve(x >> 2 == 3)
solve(x << 2 == 3)
solve(x << 2 == 24)
```

Uninterpreted Functions

```
x=Int('x')
y=Int('y')
f = Function('f', IntSort(), IntSort())
s = Solver()
s.add(f(f(x)) == x, f(x) == y, x != y)
print (s.check())
m = s.model()
print (m)
print ("f(f(x)) =", m.evaluate(f(f(x))))
print ("f(x) =", m.evaluate(f(x)))
```

Program Equivalence with Uninterpreted Functions

- Suppose we want to prove equivalence of the following two programs:

```
int fun1(int y) {
  int x, z, w;
  z = y;
  w = x;
  x = z;
  return x*x;
}
```

```
int fun2(int y) {
 return y*y;
}
```

- Let r_1, r_2 be return values of `fun1` and `fun2` respectively.
- We want to prove unsatisfiability of

$$z = y \wedge w = x \wedge x = z \wedge r_1 = x \times x \wedge r_2 = y \times y \wedge \neg(r_1 = r_2)$$

Program Equivalence with Uninterpreted Functions

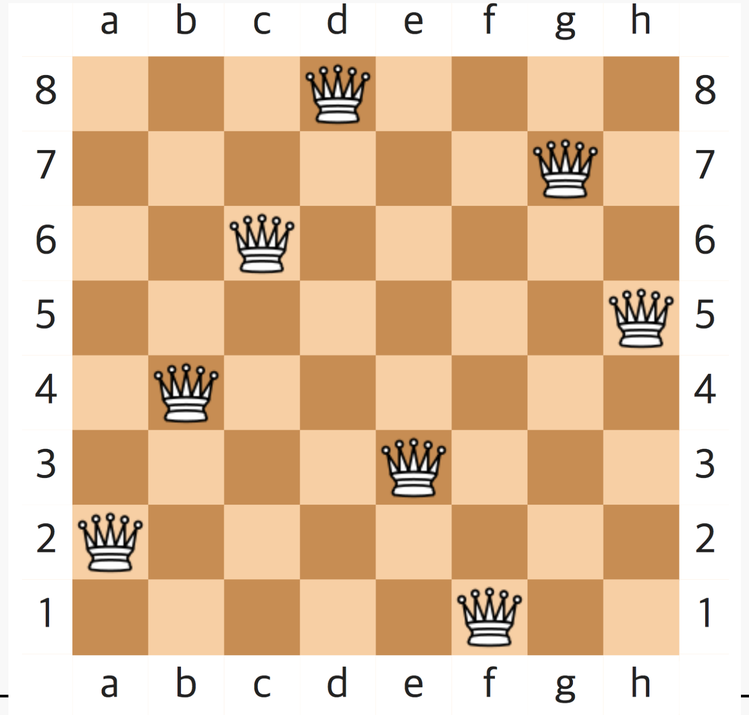
- Using an uninterpreted function *sqr*, we can rewrite the formula as

$$z = y \wedge w = x \wedge x = z \wedge r_1 = sqr(x) \wedge r_2 = sqr(y) \wedge \neg(r_1 = r_2)$$

which is UNSAT in the theory of equality with uninterpreted functions.

Eight Queens

- The eight queens puzzle is the problem of placing eight chess 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.



Encoding for Eight Queens

- Define variables Q_i : the column position of the queen in row i
- Each queen is in a column {1, …, 8}:

$$\bigwedge_{i=1}^{8} 1 \leq Q_i \wedge Q_i \leq 8$$

- No queens share the same column:

$$\bigwedge_{i=1}^{8} \bigwedge_{j=1}^{8} i \neq j \implies Q_i \neq Q_j$$

- No queens share the same diagonal:

$$\bigwedge_{i=1}^{8} \bigwedge_{j=1}^{8} i \neq j \implies Q_i - Q_j \neq i - j \wedge Q_i - Q_j \neq j - i$$

Encoding for Eight Queens

```
# 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) ]

# Each queen is in a column {1, ... 8 }
val_c = [ And(1 <= Q[i], Q[i] <= 8) for i in range(8) ]

# At most one queen per column
col_c = [ Distinct(Q) ]

# Diagonal constraint
diag_c = [ If(i == j,
              True,
              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)
```

Bit Tricks

- Low level hacks (http://graphics.stanford.edu/~seander/bithacks.html) are very popular with C programmers.
- **Power of two**: this hack is frequently used in C programs to test whether a machine integer is a power of two. We can use Z3 to prove it really works. The claim is that `x != 0 && !(x & (x - 1))` is true if and only if `x` is a power of two.

```
def prove(f):
    s = Solver()
    s.add(Not(f))
    if s.check() == unsat:
        print ("proved")
    else:
        print ("failed to prove")
```

```
x      = BitVec('x', 32)
powers = [ 2**i for i in range(32) ]
fast   = And(x != 0, x & (x - 1) == 0)
slow   = Or([ x == p for p in powers ])
print (fast)
prove(fast == slow)

print ("trying to prove buggy version...")
fast   = x & (x - 1) == 0
prove(fast == slow)
```

Bit Tricks

- Opposite signs: The following simple hack can be used to test whether two machine integers have opposite signs.

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

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

# Naive way to check if x and y have opposite signs
opposite = Or(And(x < 0, y >= 0),
              And(x >= 0, y < 0))

prove(trick == opposite)
```

Sudoku

- Insert the numbers in the 9 × 9 board so that each row, column, and 3 × 3 boxes must contain digits 1 through 9 exactly once.

	8	2			5			
			6			2		
6					1			
5								
			4		2			
								6
			8					5
		8			9			
			5			4	3	

Encoding for Sudoku

```
# 9x9 matrix of integer variables
X = [ [ Int("x_%s_%s" % (i+1, j+1)) for j in range(9) ]
      for i in range(9) ]

# each cell contains a value in {1, ..., 9}
cells_c  = [ And(1 <= X[i][j], X[i][j] <= 9)
             for i in range(9) for j in range(9) ]

# each row contains a digit at most once
rows_c   = [ Distinct(X[i]) for i in range(9) ]

# each column contains a digit at most once
cols_c   = [ Distinct([ X[i][j] for i in range(9) ])
             for j in range(9) ]

# each 3x3 square contains a digit at most once
sq_c     = [ Distinct([ X[3*i0 + i][3*j0 + j]
                        for i in range(3) for j in range(3) ])
             for i0 in range(3) for j0 in range(3) ]

sudoku_c = cells_c + rows_c + cols_c + sq_c
```

```
# sudoku instance, we use '0' for empty cells
instance = ((0,0,0,0,9,4,0,3,0),
            (0,0,0,5,1,0,0,0,7),
            (0,8,9,0,0,0,0,4,0),
            (0,0,0,0,0,0,2,0,8),
            (0,6,0,2,0,1,0,5,0),
            (1,0,2,0,0,0,0,0,0),
            (0,7,0,0,0,0,5,2,0),
            (9,0,0,0,6,5,0,0,0),
            (0,4,0,9,7,0,0,0,0))

instance_c = [ If(instance[i][j] == 0,
                  True,
                  X[i][j] == instance[i][j])
               for i in range(9) for j in range(9) ]

s = Solver()
s.add(sudoku_c + instance_c)
if s.check() == sat:
    m = s.model()
    r = [ [ m.evaluate(X[i][j]) for j in range(9) ]
          for i in range(9) ]
    print_matrix(r)
else:
    print ("failed to solve")
```