AAA615: Formal Methods

Lecture 4 — Problem-Solving with SMT Solvers

Hakjoo Oh
2017 Fall

The Z3 SMT Solver

- A popular SMT solver from Microsoft Research:

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

- Supported theories:
  - Propositional Logic
  - Theory of Equality
  - Uninterpreted Functions
  - Arithmetic
  - Arrays
  - Bit-vectors, ...
- References
  - Z3 Guide
    `https://rise4fun.com/z3/tutorialcontent/guide`
  - Z3 API in Python
    `http://ericpony.github.io/z3py-tutorial/guide-examples.htm`

Propositional Logic

```
p = Bool('p')
q = Bool('q')
r = Bool('r')
solve(Implies(p, q), r == Not(q), Or(Not(p), r))
```

```
[q = False, p = False, r = True]
```

Arithmetic

```python
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)
```

```
$ python test.py
[y = 0, x = 7]
[x = 1/8, y = 2]
```

BitVectors

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

solve(x & y == ~y)
solve(x >> 2 == 3)
solve(x << 2 == 3)
solve(x << 2 == 24)
```

```
[y = 4294967295, x = 0]
[x = 12]
no solution
[x = 6]
```

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))
```

```
sat
[x = 0, y = 1, f = [0 -> 1, 1 -> 0, else -> 1]]
f(f(x)) = 0
f(x)    = 1
```

Constraint Generation with Python List

```
X = [ Int('x%s' % i) for i in range(5) ]
Y = [ Int('y%s' % i) for i in range(5) ]
print X, Y
X_plus_Y = [ X[i] + Y[i] for i in range(5) ]
X_gt_Y = [ X[i] > Y[i] for i in range(5) ]
print X_plus_Y
print X_gt_Y
a = And(X_gt_Y)
print a
solve(a)
```

```
[x0, x1, x2, x3, x4] [y0, y1, y2, y3, y4]
[x0 + y0, x1 + y1, x2 + y2, x3 + y3, x4 + y4]
[x0 > y0, x1 > y1, x2 > y2, x3 > y3, x4 > y4]
And(x0 > y0, x1 > y1, x2 > y2, x3 > y3, x4 > y4)
[y4 = 0, x4 = 1, y3 = 0, x3 = 1, y2 = 0, x2 = 1,
 y1 = 0, x1 = 1, y0 = 0, x0 = 1]
```

Problem 1: Program Equivalence

Consider the two code fragments.

```
if (!a&&!b) then h
else if (!a) then g else f
```

```
if (a) then f
else if (b) then g else h
```

The latter might have been generated from an optimizing compiler. We would like to prove that the two programs are equivalent.

Encoding in Propositional Logic

The if-then-else construct can be replaced by a PL formula as follows:

$$\text{if } x \text{ then } y \text{ else } z \equiv (x \land y) \lor (\neg x \land z)$$

The problem of checking the equivalence is to check the validity of the formula:

$$\begin{aligned} F : & (\neg a \land \neg b) \land h \lor \neg(\neg a \land \neg b) \land (\neg a \land g \lor a \land f) \\ \iff & \; a \land f \lor \neg a \land (b \land g \lor \neg b \land h) \end{aligned}$$

If $\neg F$ is unsatisfiable, the two expressions are equivalent. Write a Python program that checks the validity of the formula F.

Problem 2: Seat Assignment

Consider three persons A, B, and C who need to be seated in a row. There are three constraints:

- A does not want to sit next to C
- A does not want to sit in the leftmost chair
- B does not want to sit to the right of C

We would like to check if there is a seat assignment for the three persons that satisfies the above constraints.

Encoding in Propositional Logic

To encode the problem, let X_{ij} be boolean variables such that

$$X_{ij} \iff \text{person } i \text{ seats in chair } j$$

We need to encode two types of constraints.

- Valid assignments:
  - Every person is seated

$$\bigwedge_i \bigvee_j X_{ij}$$

  - Every seat is occupied

$$\bigwedge_j \bigvee_i X_{ij}$$

  - One person per seat

$$\bigwedge_{i,j} (X_{ij} \implies \bigwedge_{k \neq j} \neg X_{ik})$$

Encoding in Propositional Logic

- Problem constraints:
  - A does not want to sit next to C:

$$(X_{00} \implies \neg X_{21}) \wedge (X_{01} \implies (\neg X_{20} \wedge \neg X_{22})) \wedge (X_{02} \implies \neg X_{21})$$

  - A does not want to sit in the leftmost chair

$$\neg X_{00}$$

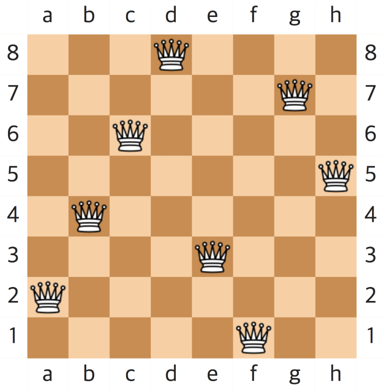
  - B does not want to sit to the right of C

$$(X_{20} \implies \neg X_{11}) \wedge (X_{21} \implies \neg X_{12})$$

Write a Python program that solves the problem.

Problem 3: 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

Define boolean variables Q_i as follows:

$$Q_i : \text{the column position of the queen in row } i$$

- Each queen is in a column $\{1, \ldots, 8\}$:

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

- 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:

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

In Python

```python
from z3 import *

def print_board (r):
  for i in range(8):
      for j in range(8):
          if r[i] == j+1:
              print 1,
          else:
              print 0,
      print ""

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) ]
col_c = [ Implies (i <> j, Q[i] <> Q[j]) for i in range(8) for j in range(8) ]
diag_c = [ Implies (i <> j, And (Q[i]-Q[j] != i-j, Q[i]-Q[j] != j-i)) for i in range(8) for j in range(i) ]
```

In Python

```
s = Solver()
s.add (val_c + col_c + diag_c)
res = s.check()
if res == sat:
   m = s.model ()
   r = [ m.evaluate (Q[i]) for i in range(8) ]
   print_board (r)
   print ""
```

```
$ python queens.py
0 0 0 1 0 0 0 0
0 1 0 0 0 0 0 0
0 0 0 0 0 0 0 1
0 0 0 0 0 1 0 0
1 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0
0 0 0 0 1 0 0 0
0 0 0 0 0 0 1 0
```

Finding All Solutions

There are multiple solutions to the eight queens problem. For example, the following can also be a solution:

```
0 0 0 0 1 0 0 0
0 0 0 0 0 0 1 0
0 0 0 1 0 0 0 0
1 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0
0 0 0 0 0 0 0 1
0 0 0 0 0 1 0 0
0 1 0 0 0 0 0 0
```

How many different solutions can you find? Write a Python program that finds all solutions of the problem.