Solver-aided reasoning

UW CSE P 504

Outline

- Primer on solver-aided reasoning
- SMTLIB and Z3
- Examples

What is a SAT solver?

What is a SAT solver?

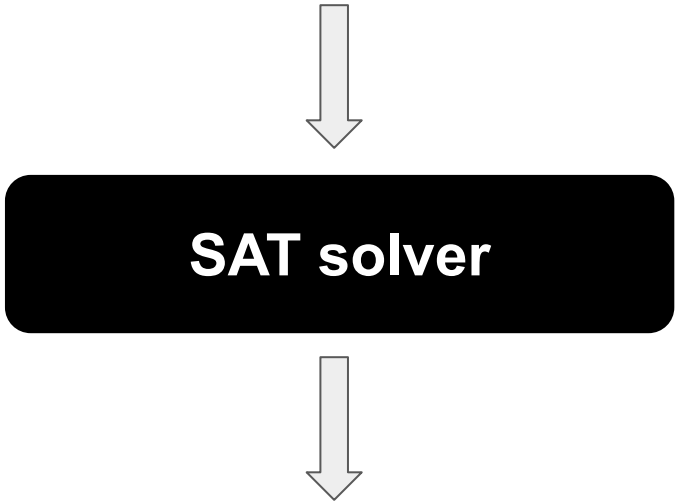
- Takes a **formula** (propositional logic) as input.

$$(X1 \vee X2) \wedge (\neg X1 \vee X3) \wedge (X1 \vee \neg X3) \wedge (\neg X2 \vee \neg X3)$$

What is a SAT solver?

- Takes a **formula** (propositional logic) as input.
- Returns a **model** (an assignment that satisfies the formula).

$$(X1 \vee X2) \wedge (\neg X1 \vee X3) \wedge (X1 \vee \neg X3) \wedge (\neg X2 \vee \neg X3)$$



$X1$ = true, $X2$ = false, $X3$ = T

Z3: an SMT solver

- SMT = Satisfiability Modulo Theories
- Input language: SMT-LIB
  - Print to the screen.
  - Declare variables and functions.

```
(echo "Running Z3...")
(declare-const a Int)
```

Z3: an SMT solver

- SMT = Satisfiability Modulo Theories
- Input language: SMT-LIB
  - Print to the screen.
  - Declare variables and functions.
  - Define constraints.

```
(echo "Running Z3...")
(declare-const a Int)
(assert (> a 0))
```

Z3: an SMT solver

- SMT = Satisfiability Modulo Theories
- Input language: SMT-LIB
  - Print to the screen.
  - Declare variables and functions.
  - Define constraints.
  - Check satisfiability and obtain a model.
  - ...

```
(echo "Running Z3...")
(declare-const a Int)
(assert (> a 0))
(check-sat)
(get-model)
```

Which question does this code answer?

Z3: an SMT solver

- SMT = Satisfiability Modulo Theories
- Input language: SMT-LIB
  - Print to the screen.
  - Declare variables and functions.
  - Define constraints.
  - Check satisfiability and obtain a model.
  - ...

```
(echo "Running Z3...")
(declare-const a Int)
(assert (> a 0))
(check-sat)
(get-model)
```

This code is asking the question:
Does an integer greater than 0 exist?

Can a+b equal a*b?

```
1 int plusEqualsMult(int a, int b) {
2   assert b > 0;
3   if (a + b == a * b) {
4     return 1;
5   }
6   return 0;
7 }
```

Does this method ever return 1?

Can a+b equal a*b?

```
1 int plusEqualsMult(int a, int b) {
2   assert b > 0;
3   if (a + b == a * b) {
4     return 1;
5   }
6   return 0;
7 }
```

```
(declare-const a Int)
(declare-const b Int)

(assert (> b 0))
(assert (= (+ a b) (* a b)))

(check-sat)
(get-model)
```

Does this method ever return 1? Let's ask Z3...

Can a+b equal a*b (but not be 0 or 4)?

```
1 int getNumber(int a, int b, int c) {
2   if (c==0) return 0;
3   if (c==4) return 0;
4   if (a + b <  c) return 1;
5   if (a + b >  c) return 2;
6   if (a * b == c) return 3;
7   return 4;
8 }
```

Does this method ever return 3?
What constraints must be satisfied?

Can a+b equal a*b (but not be 0 or 4)?

```
1 int getNumber(int a, int b, int c) {
2   if (c==0) return 0;
3   if (c==4) return 0;
4   if (a + b <  c) return 1;
5   if (a + b >  c) return 2;
6   if (a * b == c) return 3;
7   return 4;
8 }
```

All of the following must be true:

- `!(c == 0)`
- `!(c == 4)`
- `!(a + b < c)`
- `!(a + b > c)`
- `a * b == c`

Does this method ever return 3?

Can a+b equal a*b (but not be 0 or 4)?

```
1 int getNumber(int a, int b, int c) {
2   if (c==0) return 0;
3   if (c==4) return 0;
4   if (a + b <  c) return 1;
5   if (a + b >  c) return 2;
6   if (a * b == c) return 3;
7   return 4;
8 }
```

All of the following must be true:

- `!(c == 0)`
- `!(c == 4)`
- `!(a + b < c)`
- `!(a + b > c)`
- `a * b == c`

$$(a + b == c) \land (a * b == c) \land (c != 0) \land (c != 4)$$

Can a+b equal a*b (but not be 0 or 4)?

```
1 int getNumber(int a, int b, int c) {
2   if (c==0) return 0;
3   if (c==4) return 0;
4   if (a + b <  c) return 1;
5   if (a + b >  c) return 2;
6   if (a * b == c) return 3;
7   return 4;
8 }
```

All of the following must be true:

- `!(c == 0)`
- `!(c == 4)`
- `!(a + b < c)`
- `!(a + b > c)`
- `a * b == c`

```
(declare-const a Int)
(declare-const b Int)
(declare-const c Int)

(assert (not (= c 0)))
(assert (not (= c 4)))
(assert (not (< (+ a b) c)))
(assert (not (> (+ a b) c)))
(assert (= (* a b) c))

(check-sat)
```

Can a+b equal a*b (but not be 0 or 4)?

```
1 int getNumber(int a, int b, int c) {
2   if (c==0) return 0;
3   if (c==4) return 0;
4   if (a + b <  c) return 1;
5   if (a + b >  c) return 2;
6   if (a * b == c) return 3;
7   return 4;
8 }
```

All of the following must be true:

- `!(c == 0)`
- `!(c == 4)`
- `!(a + b < c)`
- `!(a + b > c)`
- `a * b == c`

Can a+b equal a*b (but not be 0 or 4)?

```
1 int getNumber(int a, int b, int c) {
2   if (c==0) return 0;
3   if (c==4) return 0;
4   if (a + b <  c) return 1;
5   if (a + b >  c) return 2;
6   if (a * b == c) return 3;
7   return 4;
8 }
```

All of the following must be true:

- `!(c == 0)`
- `!(c == 4)`
- `!(a + b < c)`
- `!(a + b > c)`
- `a * b == c`

There is no solution among the integers.
When run on your computer, can this routine return 3?

Can a+b equal a*b (but not be 0 or 4)?

```
1 int getNumber(int a, int b, int c) {
2   if (c==0) return 0;
3   if (c==4) return 0;
4   if (a + b <  c) return 1;
5   if (a + b >  c) return 2;
6   if (a * b == c) return 3;
7   return 4;
8 }
```

All of the following must be true:

- `!(c == 0)`
- `!(c == 4)`
- `!(a + b < c)`
- `!(a + b > c)`
- `a * b == c`

Z3 supports bitvectors of arbitrary size.
Let's model 32-bit ints and ask again.

Can a+b equal a*b (but not be 0 or 4)?

```
1 int getNumber(int a, int b, int c) {
2   if (c==0) return 0;
3   if (c==4) return 0;
4   if (a + b <  c) return 1;
5   if (a + b >  c) return 2;
6   if (a * b == c) return 3;
7   return 4;
8 }
```

All of the following must be true:

- `!(c == 0)`
- `!(c == 4)`
- `!(a + b < c)`
- `!(a + b > c)`
- `a * b == c`

```
(define-sort Int32 () (_ BitVec 32))

(declare-const a Int32)
(declare-const b Int32)
(declare-const c Int32)

(assert (not (= c #x00000000)))
(assert (not (= c #x00000004)))
(assert (not (bvslt (bvadd a b) c)))
(assert (not (bvsgt (bvadd a b) c)))
(assert (= (bvmul a b) c))

(check-sat)
(get-model)
```

Reasoning about program equivalence

```
1 int add(int a, int b) {
2   return a + b;
3 }
4
5 int mul(int a, int b) {
6   return a * b;
7 }
```

Are these two procedures semantically equivalent?

Reasoning about program equivalence

```
1 int add(int a, int b) {
2   return a + b;
3 }
4
5 int mul(int a, int b) {
6   return a * b;
7 }
```

```
(declare-const a Int)
(declare-const b Int)

(declare-const add Int)
(declare-const mul Int)

(assert (= add (+ a b)))
(assert (= mul (* a b)))
(assert (= add mul))

(check-sat)
(get-model)
```

Are these two procedures semantically equivalent?

Reasoning about program equivalence

```
1 int add(int a, int b) {
2   return a + b;
3 }
4
5 int mul(int a, int b) {
6   return a * b;
7 }
```

```
(declare-const a Int)
(declare-const b Int)

(declare-const add Int)
(declare-const mul Int)

(assert (= add (+ a b)))
(assert (= mul (* a b)))
(assert (= add mul))

(check-sat)
(get-model)
```

Yes, for a=2 and b=2.

What have we actually proven here?

Reasoning about program equivalence

```
1 int add(int a, int b) {
2   return a + b;
3 }
4
5 int mul(int a, int b) {
6   return a * b;
7 }
```

```
(declare-const a Int)
(declare-const b Int)

(declare-const add Int)
(declare-const mul Int)

(assert (= add (+ a b)))
(assert (= mul (* a b)))
(assert (= add mul))

(check-sat)
(get-model)
```

For **universal claims**, our goal is to **prove** the absence of counter examples (i.e., the defined constraints are **`unsat`**)!

An alternative to 64-bit xor

My  key was broken.

I had to code like this:

```
float avogadro = 5.02e+23 + 1e+23
```

But then I needed to do xor (exclusive or).

Instead of `x ^ y`, I wrote: `x + y - 2 * (x & y)`

Was I right?

Claim: `x ^ y = x + y - 2 * (x & y)`

```
(declare-const x (_ BitVec 64))
(declare-const y (_ BitVec 64))

;; (x + y) - ((x & y) << 1)
(assert (not (= (bvxor x y)
                (bvsub
                   (bvadd x y)
                   (bvshl (bvand x y) (_ bv1 64))))))

(check-sat)
```

Summary

- Solver-aided reasoning is used for testing and verification.
- SMT solvers:
  - Provide one solution, if one exists.
  - Are commonly used to find counter-examples (or prove `unsat`).
  - Support many theories that can model program semantics.
  - Usually support a standard language (SMT-lib).
- The challenge is to model a problem as a constraint system.

  A few examples:
  - Statistical test selection
  - Data-structure synthesis
  - Program synthesis
- Many higher-level DSLs and language bindings exist.

In-class 7: formal methods