CSE 403
Software Engineering
Winter 2023

Advanced program analysis
A primer on solver-aided reasoning and verification
What is a SAT solver?
What is a SAT solver?

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

\[(X_1 \lor x_2) \land (\neg x_1 \lor x_3) \land (x_1 \lor \neg x_3) \land (\neg x_2 \lor \neg x_3)\]
What is a SAT solver?

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

\[(X_1 \lor x_2) \land (\neg x_1 \lor x_3) \land (x_1 \lor \neg x_3) \land (\neg x_2 \lor \neg x_3)\]

\[X = \{X_1, X_2, X_3\} = \{T, F, T\}\]
What is Z3?

- An SMT (Satisfiability Modulo Theories) solver.
  - Supports formulas for more complex data types
  - Theories for Integers, Strings, Arrays, etc.
What is Z3?

- An SMT (Satisfiability Modulo Theories) solver.
  - Supports formulas for more complex data types
  - Theories for Integers, Strings, Arrays, etc.
  - Examples for Integers:
    - \(a \times 1 = a\) (identity element)
    - \(a + 0 = a\) (identity element)
What is Z3?

- An SMT (Satisfiability Modulo Theories) solver.
- Uses a standard language (SMT-LIB).
  - Print to the screen.
  - Declare variables and functions.

(echo "Running Z3...")
(declare-const a Int)
What is Z3?

- An SMT (Satisfiability Modulo Theories) solver.
- Uses a standard language (SMT-LIB).
  - Print to the screen.
  - Declare variables and functions.
  - Define constraints.

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

- An SMT (Satisfiability Modulo Theories) solver.
- Uses a standard language (SMT-LIB).
  - Print to the screen.
  - Declare variables and functions.
  - Define constraints.
  - Check satisfiability and obtain a model.
  - ...

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

Which question does this code answer?
What is Z3?

- An SMT (Satisfiability Modulo Theories) solver.
- Uses a standard language (SMT-LIB).
  - Print to the screen.
  - Declare variables and functions.
  - Define constraints.
  - Check satisfiability and obtain a model.
  - ...

```lisp
(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?
A first example

```c
int simpleMath(int a, int b) {
    assert(b>0);
    if(a + b == a * b) {
        return 1;
    }
    return 0;
}
```

Does this method ever return 1?
A first example

```c
int simpleMath(int a, int b) {
    assert(b>0);
    if(a + b == a * b) {
        return 1;
    }
    return 0;
}
```

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

```
(declare-const a Int)
(declare-const b Int)
(assert (> b 0))
(assert (= (+ a b) (* a b)))
(check-sat)
(get-model)
```
A more complex example

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

Does this method ever return 3?
What constraints must be satisfied?
A more complex example

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

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?
A more complex example

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

All of the following must be true:

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

\[(a + b == c) \land (a * b == c) \land (c != 0) \land (c != 4)\]
A more complex example

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

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)
A more complex example

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

All of the following must be true:

- !(c == 0)
- !(c == 4)
- !(a + b < c)
- !(a + b > c)
- a * b == c
A more complex example

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

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 Java ints (32 bits) and ask the same question...
A more complex example

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

All of the following must be true:

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

```
(define-sort JInt () (_ BitVec 32))
declare-const a JInt
declare-const b JInt
declare-const c JInt
(assert (not (= c #x00000000)))
(assert (not (= c #x00000004)))
(assert (not (bvslt (bvadd a b) c)))
(assert (not (bvsigt (bvadd a b) c)))
(assert (= (bvmul a b) c))
(check-sat)
(get-model)
```
Reasoning about program equivalence

```c
int add1(int a, int b) {
    return a + b;
}
```

```c
int add2(int a, int b) {
    return a * b;
}
```

Are these two methods semantically equivalent?
Reasoning about program equivalence

Are these two methods semantically equivalent?
Reasoning about program equivalence

```
1 int add1(int a, int b) {
2    return a + b;
3 }

4 int add2(int a, int b) {
5    return a * b;
6 }
```

(declare-const a Int)
(declare-const b Int)
(declare-const add1 Int)
(declare-const add2 Int)
(assert (= add1 (+ a b)))
(assert (= add2 (* a b)))
(assert (= add1 add2))
(check-sat)
(get-model)

Yes, for a=2 and b=2. What have we actually proven here?
Reasoning about program equivalence

```c
int add1(int a, int b) {
    return a + b;
}

int add2(int a, int b) {
    return a * b;
}
```

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