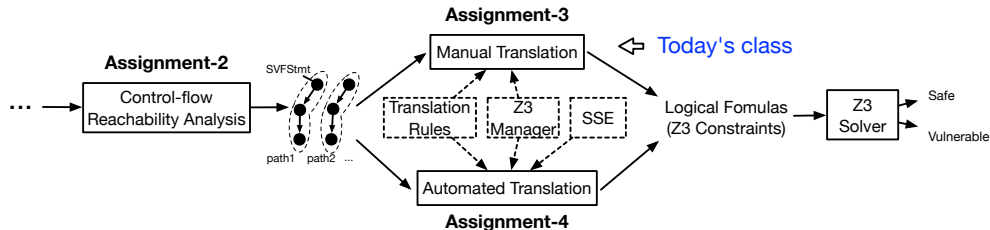


# Software Verification and Z3 Theorem Prover

Yulei Sui

University of Technology Sydney, Australia

# Today's class



- In this class, we will learn how to manually translate source code into logical formulas (Z3 constraints/expressions).
- We introduce Z3 solver, Z3 constraint format **Z3 manager** APIs.
- Then, we will demonstrate **examples** for **manual translation** from code to Z3 constraints.

# Z3 Theorem Prover

- Z3 is a Satisfiability Modulo Theories (SMT) solver from Microsoft Research<sup>1</sup>.
- Targeted at solving problems in software verification and software analysis.
- Main applications are static checking, test case generation, and more ..



Hardware verification



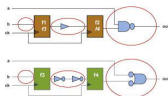
Software analysis/testing



Architecture



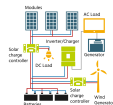
Modeling



Geometrical solving



Biological analysis

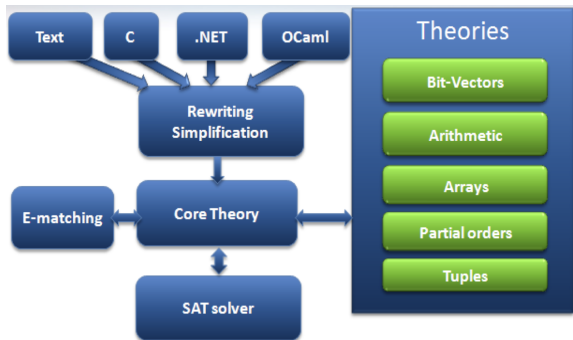


Hybrid system analysis

...

<sup>1</sup><http://research.microsoft.com/projects/z3>

# Z3 Framework<sup>2</sup>



- Z3 is an effective tool to solve **logical formulas** (Z3 constraints).
- <https://github.com/Z3Prover/z3>.
- Its SMT solver supports theories such as fixed-size bit-vectors, arithmetic, extensional arrays, datatypes, uninterpreted functions, and quantifiers.
- Z3 has official APIs for **C**, **C++**, **Python**, **.NET**, etc.
- **Z3 solver** can find one of the feasible solutions in a set of constraints.

<sup>2</sup>[https://nikolajbjorner.github.io/slides/Z3\\_System.pdf](https://nikolajbjorner.github.io/slides/Z3_System.pdf)

## Z3 Learning Materials

- Z3 GitHub repository <https://github.com/z3prover/z3>
- Getting Started with Z3: A Guide <https://jfmc.github.io/z3-play>
- Z3 tutorials [https://github.com/philzook58/z3\\_tutorial](https://github.com/philzook58/z3_tutorial)
- Z3 slides <https://github.com/Z3Prover/z3/wiki/Slides>
- Programming Z3 <http://theory.stanford.edu/~nikolaj/programmingz3.html#sec-logical-interface>

# Z3 Solver and Z3 Formulas

Z3 solver accepts a first-order (predicate) logical formula  $\phi$ , and outputs one of the following results.

- **sat** if  $\phi$  is satisfiable
- **unsat** if there is a counterexample which make  $\phi$  unsatisfiable
- **unknown** if  $\phi$  is too complex and can not be solved within a time frame.

# Z3 Solver and Z3 Formulas

Z3 solver accepts a first-order (predicate) logical formula  $\phi$ , and outputs one of the following results.

- **sat** if  $\phi$  is satisfiable
- **unsat** if there is a counterexample which make  $\phi$  unsatisfiable
- **unknown** if  $\phi$  is too complex and can not be solved within a time frame.

You play around and check the satisfiability of your Z3 constraints/formulas here:

<https://jfmc.github.io/z3-play> or

<https://compsys-tools.ens-lyon.fr/z3/index.php>

## Z3's Logical Formula (Constants, Check-Sat and Evaluation)

The Z3 input format (formula format) is an extension of the SMT-LIB 2.0 standard<sup>3</sup>.

A Z3 formula expression (`z3::expr`) has the following keywords:

- `echo` displays a message
- `declare-const` declares a constant of a given type (a.k.a sort)
- `declare-fun` declares a function
- `assert` adds a formula into the Z3 internal stack
- `check-sat` determines whether the current formulas on the Z3 stack are satisfiable or not
- `get-model` is used to retrieve an interpretation (one solution) that makes all formulas on the Z3 internal stack true
- `eval` evaluates a variable/expression produced by a model when the formulas is satisfiable.

---

<sup>3</sup><https://homepage.cs.uiowa.edu/~tinelli/papers/BarST-SMT-10.pdf>



# Constants, Check-Sat and Evaluation (Example)

$$\phi : (x > 10) \wedge (y \equiv x + 1)$$

How to represent this formula in Z3 and feed it into Z3's solver?

# Constants, Check-Sat and Evaluation (Example)

$$\phi : (x > 10) \wedge (y \equiv x + 1)$$

How to represent this formula in Z3 and feed it into Z3's solver?

```
1 (echo "starting Z3...")
2 (declare-const x Int) /// Declare an Int type variable "x"
3 (declare-const y Int) /// Declare an Int type variable "y"
4 (assert (> x 10)) /// Add the first part (x>10) of the conjunction into the solver
5 (assert (= y (+ x 1))) /// Add the second part (y==x+1) of the conjunction
6 (check-sat) /// Check whether added formulas are satisfiable.
7 (eval x) /// Evaluate the value of x when the formula is satisfiable
8 (eval y) /// Evaluate the value of y when the formula is satisfiable
```

# Constants, Check-Sat and Evaluation (Example)

$$\phi : (x > 10) \wedge (y \equiv x + 1)$$

How to represent this formula in Z3 and feed it into Z3's solver?

```
1 (echo "starting Z3...")
2 (declare-const x Int) /// Declare an Int type variable "x"
3 (declare-const y Int) /// Declare an Int type variable "y"
4 (assert (> x 10)) /// Add the first part (x>10) of the conjunction into the solver
5 (assert (= y (+ x 1))) /// Add the second part (y==x+1) of the conjunction
6 (check-sat) /// Check whether added formulas are satisfiable.
7 (eval x) /// Evaluate the value of x when the formula is satisfiable
8 (eval y) /// Evaluate the value of y when the formula is satisfiable
```

Outputs of Z3's solver:

```
1 starting Z3...
2 sat /// (check-sat) result
3 11 /// the value of x as one satisfiable solution
4 12 /// the value of y as one satisfiable solution
```

# Z3's Logical Formula (Uninterpreted Function)

The basic building blocks of SMT formulas are constants and uninterpreted functions.

- An uninterpreted function **has no other property** (no priori interpretation) **than its signature** (i.e., function name and arguments).
- An uninterpreted functions in first-order logic have **no side-effects** (e.g., can not change argument values and never return different values for the same input)
- **Constants** in Z3 can also be seen as **functions that take no arguments**.
- **The details and characteristics** of uninterpreted functions are **ignored**. This can **generalize and simplify** theorems and proofs.

# Uninterpreted Function (Example)

```
1 (declare-fun f (Int) Int)    /// Function f accepts an Int argument and returns a Int
2 (assert (= (f 10) 1))      /// f(10) = 1
3 (check-sat)
```

## Uninterpreted Function (Example)

```
1 (declare-fun f (Int) Int)    /// Function f accepts an Int argument and returns a Int
2 (assert (= (f 10) 1))      /// f(10) = 1
3 (check-sat)
```

Outputs of Z3's solver:

```
1 sat
```

The solver returns `sat`, because `f` is an uninterpreted function (i.e., all that is known about `f` is its signature), so it is possible that  $f(10) = 1$ .

## Uninterpreted Function (Example)

```
1 (declare-fun f (Int) Int)    /// Function f accepts an Int argument and returns a Int
2 (assert (= (f 10) 1))      /// f(10) = 1
3 (check-sat)
```

Outputs of Z3's solver:

```
1 sat
```

The solver returns sat, because  $f$  is an uninterpreted function (i.e., all that is known about  $f$  is its signature), so it is possible that  $f(10) = 1$ .

```
1 (declare-fun f (Int) Int)    /// Function f accepts an Int argument and returns a Int
2 (assert (= (f 10) 1))      /// f(10) = 1
3 (assert (= (f 10) 2))      /// f(10) = 2
4 (check-sat)
```

## Uninterpreted Function (Example)

```
1 (declare-fun f (Int) Int)    /// Function f accepts an Int argument and returns a Int
2 (assert (= (f 10) 1))      /// f(10) = 1
3 (check-sat)
```

Outputs of Z3's solver:

```
1 sat
```

The solver returns `sat`, because `f` is an uninterpreted function (i.e., all that is known about `f` is its signature), so it is possible that  $f(10) = 1$ .

```
1 (declare-fun f (Int) Int)    /// Function f accepts an Int argument and returns a Int
2 (assert (= (f 10) 1))      /// f(10) = 1
3 (assert (= (f 10) 2))      /// f(10) = 2
4 (check-sat)
```

Outputs of Z3's solver:

```
1 unsat
```

The solver returns `unsat`, because `f`, as an uninterpreted function, can never return different values for the same input.



# Uninterpreted Function (Example)

$$\phi : f(x) \equiv f(y) \wedge x \neq y$$

```
1 (declare-const x Int)
2 (declare-const y Int)
3 (declare-fun f (Int) Int) /// Function f accepts an Int argument and returns a Int
4 (assert (= (f x) (f y)))
5 (assert (not (= x y)))
6 (check-sat)
```

# Uninterpreted Function (Example)

$$\phi : f(x) \equiv f(y) \wedge x \neq y$$

```
1 (declare-const x Int)
2 (declare-const y Int)
3 (declare-fun f (Int) Int) /// Function f accepts an Int argument and returns a Int
4 (assert (= (f x) (f y)))
5 (assert (not (= x y)))
6 (check-sat)
```

Outputs of Z3's solver:

```
1 sat
```

An uninterpreted function can have different inputs and return the same output. For example,  $f$  can always return 1 regardless the value of the input argument.

# Constants as Uninterpreted Function (Example)

$$\phi : (x > 10) \wedge (y \equiv x + 1)$$

```
1 (declare-fun x () Int) /// "x" and "y" as an uninterpreted functions
2 (declare-fun y () Int) /// Accepts no argument and return an Int
3 (assert (> x 10))
4 (assert (= y (+ x 1)))
5 (check-sat)
6 (get-model)
```

Outputs of Z3's solver:

```
1 sat
2 (
3   (define-fun x () Int
4     11)          /// x is evaluated to be 11 for this model
5   (define-fun y () Int
6     12)          /// y is evaluated to be 11 for this model
7 )
```

(declare-const x Int) can be seen as the syntax sugar for (declare-fun x () Int).

## Z3's Logical Formula (Arithmetic)

- Z3 supports majority of commonly used arithmetic operators, such as +, -, \*, /, <<, >>, <, >, &, | (The ones listed in SVFIR)
- Types of any two operands should be the same otherwise a type conversion is needed.
- Never mix types in arithmetic, and always be explicit.

```
1 (declare-const a Int)
2 (declare-const b Float32)
3 (assert (= a (+ b 1)))
4 (check-sat)
```

Outputs of Z3's solver:

```
1 Error: (error "line 3 column 19: Sort mismatch at argument #1 for function
2 (declare-fun + (Int Int) Int) supplied sort is (_ FloatingPoint 8 24)")
```

## Z3's Logical Formula (if-then-else Expression)

- `ite(b, x, y)` represents a conditional expression, where `b` is the condition, `ite` returns `x` if `b` is evaluated true, otherwise `y` is returned
- Used for comparison or branches

```
1 (ite (and (= x!1 11) (= x!2 false)) 21 0)
```

The above Z3 formula evaluates (returns) 21 when `x!1` is equal to 11, and `x!2` is equal to false. Otherwise, it returns 0.

## Z3's Logical Formula (Arrays)

Formulating a program of a mathematical theory of computation McCarthy proposed a basic theory of arrays as characterized by the **select-store** axioms.

- `(select a i)`: returns the value stored at position `i` of the array `a`;
- `(store a i v)`: returns a new array identical to `a`, but on position `i` it contains the value `v`.
- Z3 assumes that arrays are extensional over `select`. Z3 also enforces that if two arrays agree on all reads, then the arrays are equal.

## Z3's Logical Formula (Arrays)

Formulating a program of a mathematical theory of computation McCarthy proposed a basic theory of arrays as characterized by the **select-store** axioms.

- `(select a i)`: returns the value stored at position `i` of the array `a`;
- `(store a i v)`: returns a new array identical to `a`, but on position `i` it contains the value `v`.
- Z3 assumes that arrays are extensional over `select`. Z3 also enforces that if two arrays agree on all reads, then the arrays are equal.

The following formulas store `y` to the `x`-th position of array `a` and then load the value at `a`'s `x`-th position to `z`

```
1 (declare-const x Int)
2 (declare-const y Bool)
3 (declare-const z Bool)
4 (declare-const a (Array Int Bool))  /// an array of Bools with Int as the indices
5 (assert (= (store a x y) a))        /// a[x] == y
6 (assert (= (select a x) z))         /// z == a[x]
```

## Z3's Logical Formula (Scopes)

Z3 maintains a global stack of declarations and assertions via **push** and **pop**

- **push**: creates a new scope by saving the current stack size.
- **pop**: removes any assertion or declaration performed between it and the matching push.

The `check-sat` command always operates on the current global stack.



## Z3's Logical Formula (Scopes)

Z3 maintains a global stack of declarations and assertions via **push** and **pop**

- **push**: creates a new scope by saving the current stack size.
- **pop**: removes any assertion or declaration performed between it and the matching push.

The check-sat command always operates on the current global stack.

```
1 (declare-const x Int)
2 (declare-const a (Array Int Int))  /// an array of Ints
3 (push)
4 (assert (= (store a 1 10) a))      /// a[1] == 10
5 (assert (= (select a 1) x))        /// x == a[1]
6 (assert (= x 20))                  /// x == 20
7 (check-sat)
8 (pop) ; remove the three assertions
9 (assert (= x 20))                  /// x == 10
10 (check-sat)
```

What is the output of the solver?

# Translating Code to Z3 Formulas

We provide a Z3Mgr class (a wrapper class to manipulate Z3 APIs) to generate Z3 formulas or so-called `z3::expr`.

API	Meanings
<code>z3::expr getZ3Expr(std::string);</code>	Create a <b>variable</b> given a string name
<code>z3::expr getZ3Expr(int);</code>	Create a <b>variable</b> given an integer
<code>z3::expr getMemObjAddress(std::string);</code>	Create a <b>memory object</b> in program
<code>z3::expr getGepObjAddress(z3::expr, u32_t);</code>	Create a <b>field object</b> with an <b>offset</b> of an aggregate
<code>void addToSolver(z3::expr);</code>	<b>Add</b> a Z3 expression/formula to the solver
<code>void resetSolver();</code>	Clean all formulas in the the solver
<code>solver.check();</code>	<b>Check satisfiability</b> of an z3 formula
<code>z3::expr getEvalExpr(z3::expr);</code>	<b>Evaluate an expression</b> based on a model
<code>void printExprValues();</code>	<b>Print</b> the values of <b>all expressions</b> in the solver.

More details, refer to

<https://github.com/SVF-tools/Teaching-Software-Verification/wiki/SVF-APIs>

# Translation Rules

expr p = getZ3Expr("p")    expr q = getZ3Expr("q")    expr r = getZ3Expr("r")		
SVFStmt	C-Like form	Operations
AddrStmt (constant)	p = c	addToSolver(p == c);
AddrStmt (mem allocation)	p = alloc	addToSolver(p == getMemObjAddress("alloc");)
CopyStmt	p = q	addToSolver(p == q);
LoadStmt	p = *q	addToSolver(p == loadValue(q));
StoreStmt	*p = q	storeValue(p, q);
GepStmt	p = &(q → i) or p = &q[i]	addToSolver(p == getGepObjAddress(q,i));
PhiStmt	r = phi(l <sub>1</sub> : p, l <sub>2</sub> : q)	if(executed from l <sub>1</sub> )    addToSolver(p==r); if(executed from l <sub>2</sub> )    addToSolver(q==r);
BranchStmt	if (p) l <sub>1</sub> else l <sub>2</sub>	expr cond = getEvalExpr(p); if(cond.is_false()) execute l <sub>2</sub> else execute l <sub>1</sub> addToSolver(cond = true);
UnaryOPStmt	¬p	addToSolver(!p);
BinaryOPStmt	r = p ⊗ q	addToSolver(r == p ⊗ q);
CmpStmt	r = p ⊙ q	addToSolver(r == ite(p ⊙ q, true, false));
CallPE/RetPE	r = f(...,q,...)    f(...,p,...){... return z}	
CallPE	p = q	solver.push();    addToSolver(p == q);
RetPE	p = r	expr ret = getZ3Expr(r); solver.pop(); addToSolver(p == ret);

# Translating Code to Z3 Formulas (Scalar Example)

The target program code needs to be in **SSA form** (e.g., SVFIR).

- Top-level variables can only be defined once
  - $a = 1; a = 2; \implies a1 = 1; a2 = 2;$
- Memory objects can only be modified/read through top-level pointers at StoreStmt and LoadStmt.
  - $p = \&a; *p = r;$  The value of  $a$  can only be modified/read via dereferencing  $p$ .

# Translating Code to Z3 Formulas (Scalar Example)

The target program code needs to be in **SSA form** (e.g., SVFIR).

- Top-level variables can only be defined once
  - $a = 1; a = 2; \implies a1 = 1; a2 = 2;$
- Memory objects can only be modified/read through top-level pointers at StoreStmt and LoadStmt.
  - $p = \&a; *p = r;$  The value of  $a$  can only be modified/read via dereferencing  $p$ .

```
int main() {  
  
    int a;  
    int b;  
    a = 0;  
    b = a + 1;  
    assert(b>0);  
  
}
```

C code

```
// int a;  
expr a = getZ3Expr("a");  
// int b;  
expr b = getZ3Expr("b");  
// a = 0;  
addToSolver(a == 0);  
// b = a+1;  
addToSolver(b == (a + 1));  
// assert(b > 0);  
addToSolver(b > 0);  
solver.check();
```

Translator

```
(declare-fun a () Int)  
(declare-fun b () Int)  
(assert (= a 0))  
(assert (= b (+ a 1)))  
(assert (> b 0))  
(check-sat)
```

Z3 Formulas

Z3's  
SMT solver

# Translating Code to Z3 Formulas (Memory Operation Example)

- Each memory object has a unique ID and allocated with a **virtual memory address**
- In our modeling, the virtual address starts from **0x7f..... + ID** (i.e., 2130706432 + ID in decimal)
- Memory operations will be through store and load values from `loc2ValMap`, an Z3 array.

```
int main() {  
    int* p;  
    int x;  
  
    p = malloc(..);  
    *p = 5;  
    x = *p;  
    assert(x==5);  
}
```

```
// int** p;  
expr p = getZ3Expr("p");  
// int x;  
expr x = getZ3Expr("x");  
// p = malloc(..);  
expr m = getMemObjAddress("malloc1");  
addToSolver(p == m);  
// *p = 5;  
storeValue(p, getZ3Expr(5));  
// x = *p;  
addToSolver(x == loadValue(p));  
// assert(x==5);  
addToSolver(x == getZ3Expr(5));  
solver.check();
```

```
(declare-fun p () Int)  
(declare-fun loc2ValMap ()  
  (Array Int Int))  
(declare-fun x () Int)  
(assert (= p 2130706435))  
(assert (= x (select  
  (store loc2ValMap 2130706435 5)  
  2130706435)))  
(assert (= x 5))  
(check-sat)
```

C code

Translator

Z3 Formulas

# What's next?

- (1) Understand Z3 formula format in the slides
- (2) Understand Z3Mgr class in the GitHub Repository of Teaching-Software-Verification
- (3) Read through Assignment-3.pdf on Canvas to understand some examples for manual code verification.
- (4) Finish the quizzes of Assignment 3 on Canvas, and implement a manual translation from code to Z3 formulas using Z3Mgr i.e., coding task in Assignment 3.