An Introduction to Z3

Huixing Fang

National Trusted Embedded Software
Engineering Technology Research Center

April 12, 2017

Outline

1. SMT
2. Z3

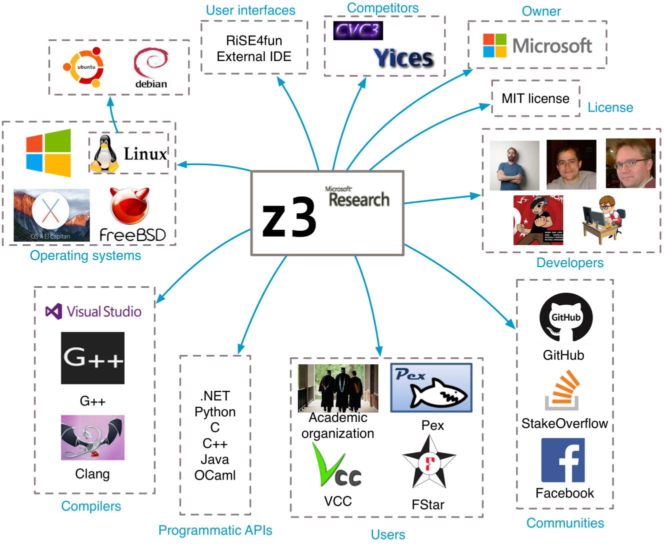
Outline

SMT: Satisfiability Modulo Theory

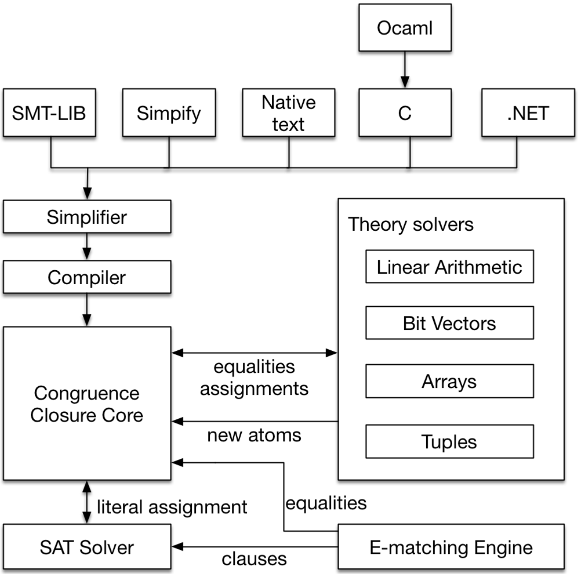
1. Satisfiability is the problem of determining whether a formula ϕ has a model
   1. If ϕ is propositional, a model is a truth assignemt to Boolean variables
   2. If ϕ is a first-order formula, a model assigns values to variables and interpretations to the function and predicate symbols
2. SAT Solvers: check satisfiability of propositional formulas
3. SMT Solvers: check satisfiability of formulas in a decidable first-order theory (e.g., linear arithmetic, array theory, bitvectors)

Outline

Z3 Context


User interfaces
RiSE4fun
External IDE
Competitors
CVC3
Yices
Owner
Microsoft
MIT license
License
ubuntu
debian
Linux
OS X El Capitan
FreeBSD
Operating systems
z3
Microsoft Research
Developers
Visual Studio
G++
G++
Clang
Compilers
.NET
Python
C
C++
Java
OCaml
Programmatic APIs
Academic organization
Pex
Pex
VCC
VCC
FStar
Users
GitHub
GitHub
StakeOverflow
Facebook
Communities

Z3 Architecture



Z3 Architecture

1. Simplifier : Input formulas are first processed using an incomplete, but efficient simplification
   - $p \wedge true \longmapsto p$
   - $x = 4 \wedge q(x) \longmapsto x = 4 \wedge q(4)$
2. Compiler: The simplified abstract syntax tree representation of the formula is converted into a different data-structure comprising of a set of clauses and congruence-closure nodes
3. Congruence Closure Core: Handles equalities and uninterpreted functions. Receives the assignment from the SAT solver, and processed using E-matching

Basic Commands

1. displays a message: (echo "starting Z3...")

Basic Commands

1. displays a message: (echo "starting Z3...")
2. declares a constant of a given type: (declare-const a Int)

Basic Commands

1. displays a message: (echo "starting Z3...")
2. declares a constant of a given type: (declare-const a Int)
3. declares a function : (declare-fun f (Int Bool) Int)

Basic Commands

1. displays a message: (echo "starting Z3...")
2. declares a constant of a given type: (declare-const a Int)
3. declares a function : (declare-fun f (Int Bool) Int)
4. asserts a formula : (assert (> a 10))

Basic Commands

1. displays a message: (echo "starting Z3...")
2. declares a constant of a given type: (declare-const a Int)
3. declares a function : (declare-fun f (Int Bool) Int)
4. asserts a formula : (assert (> a 10))
5. defines a function: (define-fun a () Int 100)

Basic Commands

1. displays a message: (echo "starting Z3...")
2. declares a constant of a given type: (declare-const a Int)
3. declares a function : (declare-fun f (Int Bool) Int)
4. asserts a formula : (assert (> a 10))
5. defines a function: (define-fun a () Int 100)
6. defines a function:
   (define-fun f ((x Int) (y Bool)) Int
   (ite (and (= x 11) (= y true)) 0 1))

Basic Commands

1. displays a message: (echo "starting Z3...")
2. declares a constant of a given type: (declare-const a Int)
3. declares a function : (declare-fun f (Int Bool) Int)
4. asserts a formula : (assert (> a 10))
5. defines a function: (define-fun a () Int 100)
6. defines a function:
   (define-fun f ((x Int) (y Bool)) Int
   (ite (and (= x 11) (= y true)) 0 1))
7. check: (check-sat) determines whether the current formulas on the Z3 stack are satisfiable or not

Basic Commands

1. displays a message: (echo "starting Z3...")
2. declares a constant of a given type: (declare-const a Int)
3. declares a function : (declare-fun f (Int Bool) Int)
4. asserts a formula : (assert (> a 10))
5. defines a function: (define-fun a () Int 100)
6. defines a function:
   (define-fun f ((x Int) (y Bool)) Int
   (ite (and (= x 11) (= y true)) 0 1))
7. check: (check-sat) determines whether the current formulas on the Z3 stack are satisfiable or not
8. model: (get-model) is used to retrieve an interpretation that makes all formulas on the Z3 internal stack true

Basic Commands

Basic Building Blocks

1. The basic building blocks of SMT formulas are constants and functions. Constants are just functions that take no arguments. So everything is really just a function.
2. An uninterpreted function or function symbol is one that has no other property than its name and n-ary form.

Basic Commands

Basic Building Blocks

1. The basic building blocks of SMT formulas are constants and functions. Constants are just functions that take no arguments. So everything is really just a function.
2. An uninterpreted function or function symbol is one that has no other property than its name and n-ary form.

Valid and Satisfiable

1. A formula F is valid if F always evaluates to true for any assignment of appropriate values to its uninterpreted function and constant symbols.
2. A formula F is satisfiable if there is some assignment of appropriate values to its uninterpreted function and constant symbols under which F evaluates to true.

Propositional Logic

The pre-defined sort Bool is the sort (type) of all Boolean propositional expressions. Z3 supports the usual Boolean operators and, or, xor, not, => (implication), ite (if-then-else).

Example

Formulas:

```
(declare-const p Bool)
(declare-const q Bool)
(declare-const r Bool)
(define-fun conjecture () Bool
            (=> (and (=> p q) (=> q r))
            (=> p r)))
(assert (not conjecture))
(check-sat)
```

Result: unsat

Uninterpreted functions and constants

The basic building blocks of SMT formulas are constants and functions. Constants are just functions that take no arguments. So everything is really just a function.

Example

Formulas:

```
(declare-fun f (Int) Int)
(declare-fun a () Int)
; a is a constant
(declare-const b Int)
; syntax sugar for (declare-fun b () Int)
(assert (> a 20))
(assert (> b a))
(assert (= (f 10) 1))
(check-sat)
(get-model)
```

Arithmetic

Z3 has builtin support for integer and real constants. These two types (sorts) represent the mathematical integers and reals.

```
(declare-const a Int)
(declare-const b Int)
(declare-const d Real)
(declare-const e Real)
(assert (> a (+ b 2)))
(assert (>= d e))
(check-sat)
(get-model)
```

```
sat
 (model
  (define-fun e () Real 0.0)
  (define-fun d () Real 0.0)
  (define-fun a () Int 1)
  (define-fun b () Int (- 2))
 )
```

Nonlinear arithmetic

1. A formula is nonlinear if it contains expressions of the form (* t s) where t and s are not numbers.
2. Nonlinear real arithmetic is very expensive, and Z3 is not complete for this kind of formula.

```
(declare-const a Int)
(assert (> (* a a) 3))
(check-sat)
(get-model)

(declare-const b Real)
(declare-const c Real)
(assert (= (+ (* b b b)
(* b c)) 3.0))
(check-sat)
```

```
sat
(model
  (define-fun a () Int
    (- 8)
  )
)

unknown
```

Bitvectors

Bitvector Literals

Bitvector literals may be defined using binary, decimal and hexadecimal notation.

1. `#b010` in binary format is a bitvector of size 3
2. bitvector literal `#x0a0` in hexadecimal format is a bitvector of size 12

The size must be specified for bitvector literals in decimal format.

- `(_ bv10 32)` is a bitvector of size 32 that representes the numeral 10

Z3 supports Bitvectors of arbitrary size. `(_ BitVec n)` is the sort of bitvectors whose length is n. Declare 64-bit bitvector constant x as follow:

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

Basic Bitvector Arithmetic

```
bvadd : addition
bvsub : subtraction
bvneg : unary minus
bvmul : multiplication
bvurem : unsigned remainder
bvsrem : signed remainder
bvsmod : signed modulo
bvshl : shift left
bvlshr : unsigned (logical) shift right
bvashr : signed (arithmetical) shift right
bvor : bitwise or
bvand : bitwise and
bvnot : bitwise not
bvnand : bitwise nand
bvnor : bitwise nor
bvxnor : bitwise xnor
```

Arrays

1. The expression (select a i) returns the value stored at position i of the array a;
2. and (store a i v) returns a new array identical to a, but on position i it contains the value v.

```
(declare-const x Int)
(declare-const y Int)
(declare-const a1 (Array Int Int))
(assert (= (select a1 x) x))
(assert (= (store a1 x y) a1))
(check-sat)
(get-model)
```

```
sat
(model
  (define-fun y () Int 1)
  (define-fun a1 () (Array Int Int) (_ as-array k!0))
  (define-fun x () Int 1)
  (define-fun k!0 ((x!1 Int)) Int (ite (= x!1 1) 1 0)) )
```

Records

A record is specified as a datatype with a single constructor and as many arguments as record elements.

Example

Parametric type Pair, with constructor mk-pair and two arguments that can be accessed using the selector functions first and second.

```
(declare-datatypes (T1 T2)
 ((Pair (mk-pair (first T1) (second T2)))))
(declare-const p1 (Pair Int Int))
(declare-const p2 (Pair Int Int))
(assert (= p1 p2))
(assert (> (second p1) 20))
(check-sat)
(get-model)
(assert (not (= (first p1) (first p2))))
(check-sat)
```

Model-based Quantifier Instantiation

The model-based quantifier instantiation (MBQI) is essentially a counter-example based refinement loop, where candidate models are built and checked.

```
(set-option :smt.mbqi true)
(declare-fun f (Int Int) Int)
(declare-const a Int)
(declare-const b Int)
(assert (forall ((x Int))
(>= (f x x) (+ x a))))
(assert (< (f a b) a))
(assert (> a 0))
(check-sat)
(get-model)

(eval (f (+ a 10) 20))
```

```
sat
  (model
  (define-fun b () Int 2)
  (define-fun a () Int 1)
  (define-fun f
    ((x!1 Int)
    (x!2 Int))
    Int
    (ite (and (= x!1 1)
              (= x!2 2))
              0
              (+ 1 x!1)))
  )

  12
```

Relations, rules and queries

The default fixed-point engine is a bottom-up Datalog engine. It works with finite relations and uses finite table representations as hash tables as the default way to represent finite relations.

```
(declare-rel a ())
(declare-rel b ())
(declare-rel c ())
(rule (=> b a))
(rule (=> c b))
(set-option :fixedpoint.engine datalog)
(query a)
(rule c)
(query a:print-answer true)
```

```
unsat
sat
true
```

Engine for Property Directed Reachability

The PDR engine is targeted at applications from symbolic model checking of software. The systems may be infinite state.

Example

McCarthy's 91 function illustrates a procedure that calls itself recursively twice

```
mc(x) = if x > 100 then x - 10 else mc(mc(x+11))
```

```
(declare-rel mc (Int Int))
(declare-var n Int)
(declare-var m Int)
(declare-var p Int)

(rule (=> (> m 100) (mc m (- m 10))))
(rule (=> (and (<= m 100) (mc (+ m 11) p) (mc p n))
          (mc m n)))
```

Engine for Property Directed Reachability

Example

McCarthy's 91 function illustrates a procedure that calls itself recursively twice

```
mc(x) = if x > 100 then x - 10 else mc(mc(x+11))
```

```
(declare-rel mc (Int Int))
(declare-var n Int)
(declare-var m Int)
(declare-var p Int)
(rule (=> (> m 100) (mc m (- m 10))))
(rule (=> (and (<= m 100) (mc (+ m 11) p) (mc p n))
          (mc m n)))

(declare-rel q (Int Int))
(rule (=> (and (mc m n) (< n 92)) (q m n)))
(query q :print-certificate true)
```

Engine for Property Directed Reachability

Example

McCarthy's 91 function illustrates a procedure that calls itself recursively twice

```
mc(x) = if x > 100 then x - 10 else mc(mc(x+11))
```

```
sat
(and
 (not (<= 92 (:var 5)))
 (mc (:var 4) (:var 5))
 (= (:var 5) (+ (- 10)
    (:var 4)))
 (not (<= (:var 4) 100))
)
```

```
v5 < 92

mc(v4) = v5

v5 =  v4 - 10

v4 > 100
```

End

Thanks for your attention.