Review

Directed Automated Random Testing [PLDI 2005]

```
int obscure(int x, int y) {
  if (x==complex(y)) error();
  return 0;
}
```

Run 1 :

- start with (random) x=33, y=42
- execute concretely and symbolically:
  if (33 != 567) | if $\boxed{\text{(x != complex(y}}$))
  constraint too complex
  → simplify it: x != 567
- solve: x==567 → solution: x=567
- new test input: x=567, y=42

Run 2 : the other branch is executed
All program paths are now covered !

Also known as concolic execution (<u>conc</u>rete + symb<u>olic</u>)
Referred to here as <u>dynamic symbolic execution</u>

Dynamic Symbolic Execution

Formula F := False

Loop

Find program input *i in solve(negate(F))* // stop if no such *i* can be found

Execute P(*i*); record path condition C // in particular, C(i) holds

F := F $\bigvee$ C

End

Lecture 2

Design and Implementation of Dynamic Symbolic Execution (for Python, in Python)

https://github.com/thomasjball/PyExZ3

The Code

- Derived from the NICE project (http://code.google.com/p/nice-of/)
- Ported to use Z3 (instead of STP)
- Removed platform dependences (should run on Linux, MacOS, etc.)
- DSE hooks solely via method overloading
  - No AST rewriting
  - No bytecode interpretation
- Made error checking more robust
- Added more regression tests

Demo

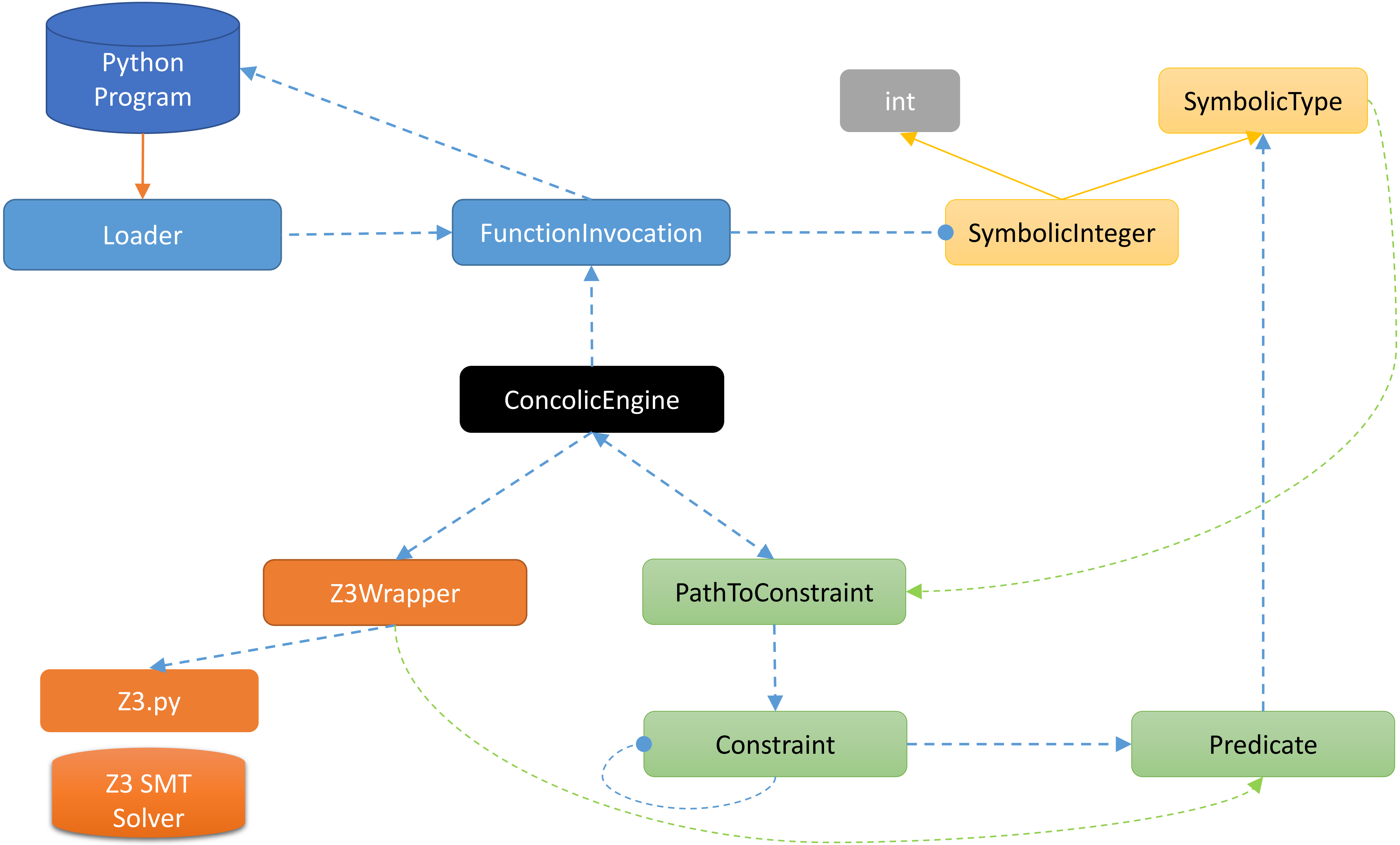
Requirements

- Identify the code under test (CUT)
- Identify symbolic inputs
- Trace the CUT
- Reinterpret instructions to compute symbolic expressions
- Collect path constraint
- Generate new input
- Restart execution of CUT (from initial state)
- Search strategy to expose new paths

Classes

- Loader
- FunctionInvocation
- SymbolicType
  - SymbolicInteger
- PathToConstraint
- Constraint
- Predicate
- Z3Wrapper
- ConcolicEngine

- Identify the code under test (CUT)
- Identify symbolic inputs
- Trace the CUT
- Reinterpret instructions
- Collect path constraint
- Generate new input from path constraint
- Restart execution of CUT (from initial state)
- Search strategy to expose new paths


Python Program
Loader
FunctionInvocation
int
SymbolicType
SymbolicInteger
ConcolicEngine
Z3Wrapper
PathToConstraint
Z3.py
Constraint
Predicate
Z3 SMT Solver

Loader

Uses reflection to

- load the code under test and identify function entry point F
- determine the number of arguments to F

- Creates a SymbolicInteger for each argument

- Creates a FunctionInvocation object to encapsulate
  - entry point F and
  - symbolic argument values

SymbolicType

SymbolicType

- An abstract base class representing a pair of
  - a concrete value of type T
  - a symbolic value of type T
- Overrides basic object operations:
  - Comparisons: __eq__, __ne__, __lt__, __le__, __gt__, __ge__
  - Coercion to Boolean: __bool__

symbolic\symbolic_types

SymbolicInteger

- A SymbolicInteger is
  - a Python int, and
  - a SymbolicType
- SymbolicInteger overloads arithmetic operations for which we know how to translate to logic and solver with Z3

Python Execution

x : SymbolicInt
- 0
- "x"

y = x + 1

y : SymbolicInt
- 1
- "x" + 1

Intercepting Control-flow in Python

- Conditionals
  - <u>if</u> e1, <u>while</u> e1, e1 <u>and</u> e2 , e1 <u>or</u> e2, <u>not</u> e

- *Any* object can be used in a conditional test
  - Python calls __bool__ method to get a Boolean from object
  - Used whenever a conditional test (predicate) encountered

- We override __bool__ in order to intercept control-flow and determine which way predicate will evaluate (true, false)

Predicate

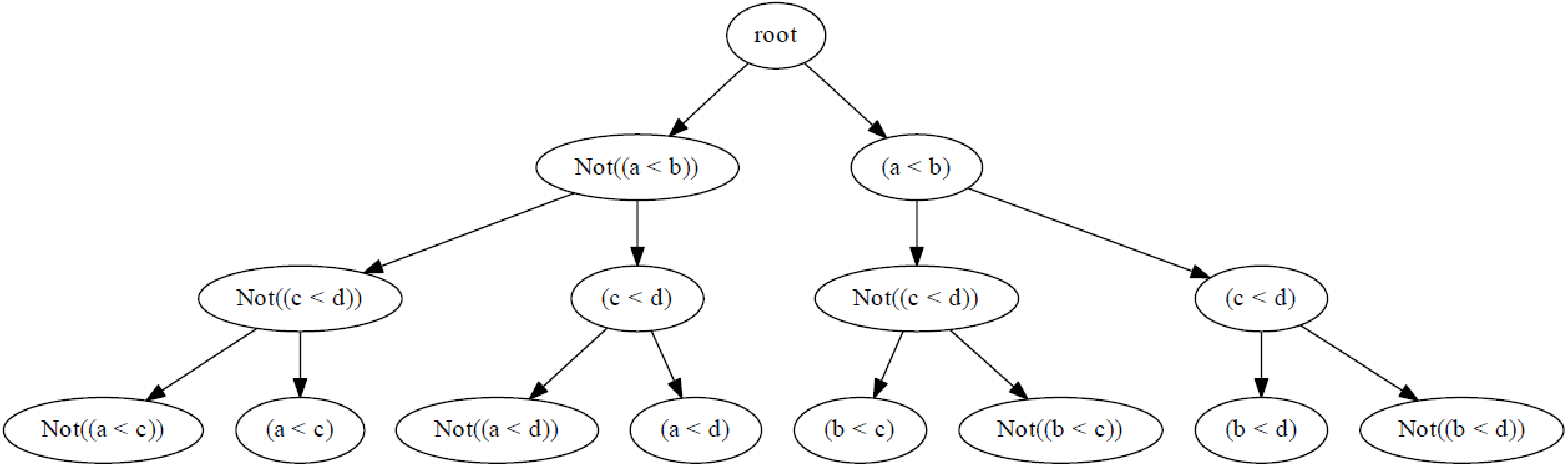
- Tracks a predicate in the program and which direction it took (T,F)

Constraint

- A sequence of predicates corresponding to an execution path

PathToConstraint

PathToConstraint



Z3Wrapper

- Translate from AST expression (in SymbolicType) into Z3 expression

Python Semantics

$\downarrow$

Logic

Python Semantics -> Logic

- Python integers
  - Python’s integer representation grows to limits of machine memory
  - Arithmetic operations do not cause overflows!

- Z3 BitVectors
  - Z3 BitVectors have finite width (they do not grow)
  - Arithmetic operations do cause overflows…

Hilbert's 10th Problem

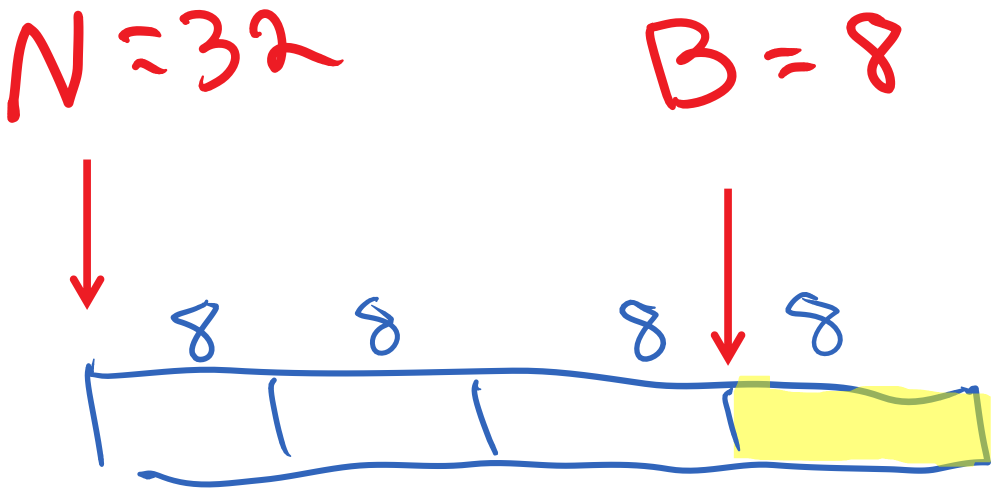
Given a Diophantine equation with any number of unknown quantities and with rational integral numerical coefficients: *To devise a process according to which it can be determined in a finite number of operations whether the equation is solvable in rational integers*.

There is a polynomial p(a,x1,…xn) with integer coefficients such that

the set of values of a for which p(a,x1,…xn) = 0

has solutions in the natural numbers is not computable.

Solution: bounded search over symbolic inputs



1. Increase B while φ_N UNSAT
2. if $m \models \varphi_N$ does $m \models \varphi$?

Demo

ConcolicEngine

- Generational search procedure

Deficiencies

- One process, many executions
  - Clean restart of state problematic
  - Means we may not be able to control program to go where we want it to…
- Poor handling of while loops
  - Take whileloop.py and remove “0 < x < 10”
  - Loop subsumption

$i = 0$

$\text{while } (i < N)$

$\qquad i = i + 1$

$$
\begin{tikzcd}
0 < N & P_0 \arrow[d, "T"'] \arrow[dr, dashed, "F"] & \\
1 < N & P_1 \arrow[d, "T"'] \arrow[dr, dashed, "F"] & {} \\
2 < N & P_2 \arrow[d, "T"'] \arrow[dr, dashed, "F"] & {} \\
3 < N & P_3 & {}
\end{tikzcd}
$$

if

$P_i \Rightarrow P_{i-1}$ then

don't flip P_i

Assignment 2

1. All PyExZ3 regression tests should pass

2. Write and submit new test cases
   - At least one test case that passes
   - At least one that shows off a deficiency in the implementation
   - Send email with new tests to tball@microsoft.com - I will add to github

3. For more fun,
   - Fix up treatment of while loops so that modified whileloop.py doesn't diverge
   - Implement loop subsumption
   - Write a function (in)equivalence checker