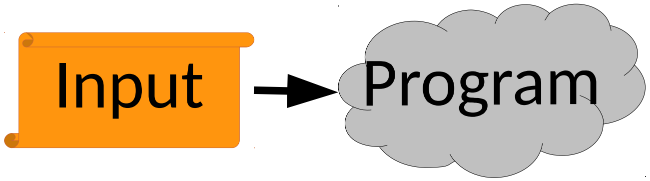
Testing, Fuzzing, & Symbolic Execution

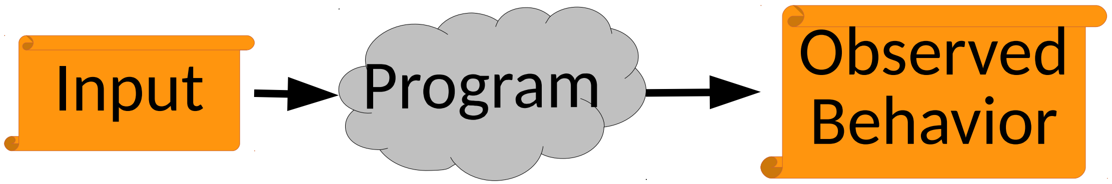
Software Testing

- The most common way of measuring & ensuring correctness



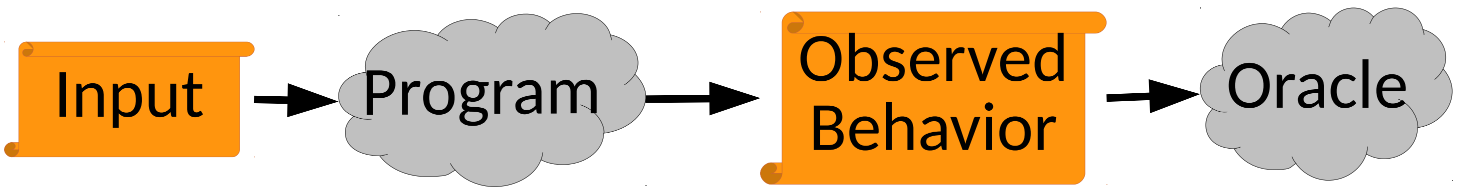
Software Testing

- The most common way of measuring & ensuring correctness



Software Testing

- The most common way of measuring & ensuring correctness

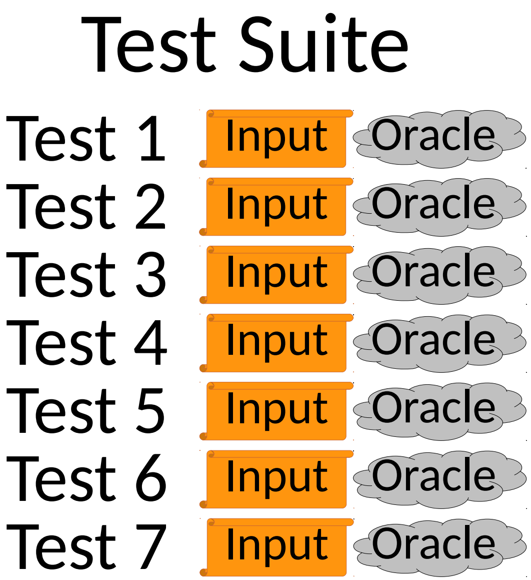


Software Testing

- The most common way of measuring & ensuring correctness

Software Testing

- The most common way of measuring & ensuring correctness



Software Testing

- The most common way of measuring & ensuring correctness

- Key Issues:
  - Are the tests adequate?

Software Testing

- The most common way of measuring & ensuring correctness

- Key Issues:
  - Are the tests adequate?
  - Automated input generation

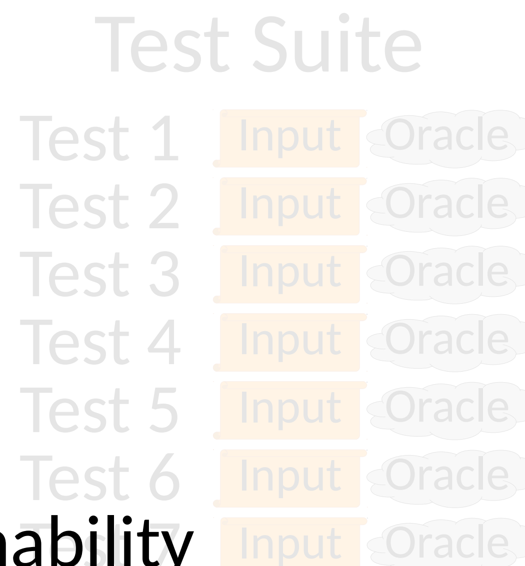
Software Testing

- The most common way of measuring & ensuring correctness

- Key Issues:
  - Are the tests adequate?
  - Automated input generation
  - Automated oracles

Software Testing

- The most common way of measuring & ensuring correctness



- Key Issues:
  - Are the tests adequate?
  - Automated input generation
  - Automated oracles
  - Robustness / flakiness / maintainability

Software Testing

- The most common way of measuring & ensuring correctness

- Key Issues:
  - Are the tests adequate?
  - Automated input generation
  - Automated oracles
  - Robustness / flakiness / maintainability
  - ...

Test Suite Adequacy

- Questions
  - Is a test suite good enough?
  - What parts of software need to be tested better?

Test Suite Adequacy

- Questions
  - Is a test suite good enough?
  - What parts of software need to be tested better?
- Metrics
  - Statement Coverage

Is each statement executed by at least one test in the test suite?

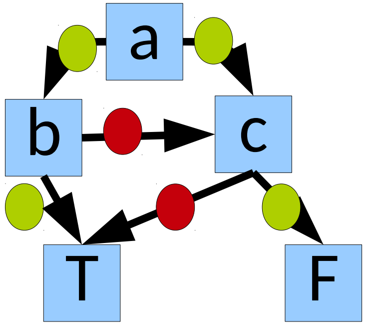
```
def my_lovely_fun(a,b,c):
  if (a && b) || c:
    …
  else:
    …
  print('awesome')
```

$$\text{score} = \frac{\text{\# covered}}{\text{\# statements}}$$

Test Suite Adequacy

- Questions
  - Is a test suite good enough?
  - What parts of software need to be tested better?
- Metrics
  - Statement Coverage
  - Branch Coverage

```
def my_lovely_fun(a,b,c):
  if (a && b) || c:
    …
  else:
    …
  print('awesome')
```



$$score = \frac{\# \text{ covered}}{\# \text{ branches}}$$

Test Suite Adequacy

- Questions
  - Is a test suite good enough?
  - What parts of software need to be tested better?
- Metrics
  - Statement Coverage
  - Branch Coverage
  - MC/DC Coverage

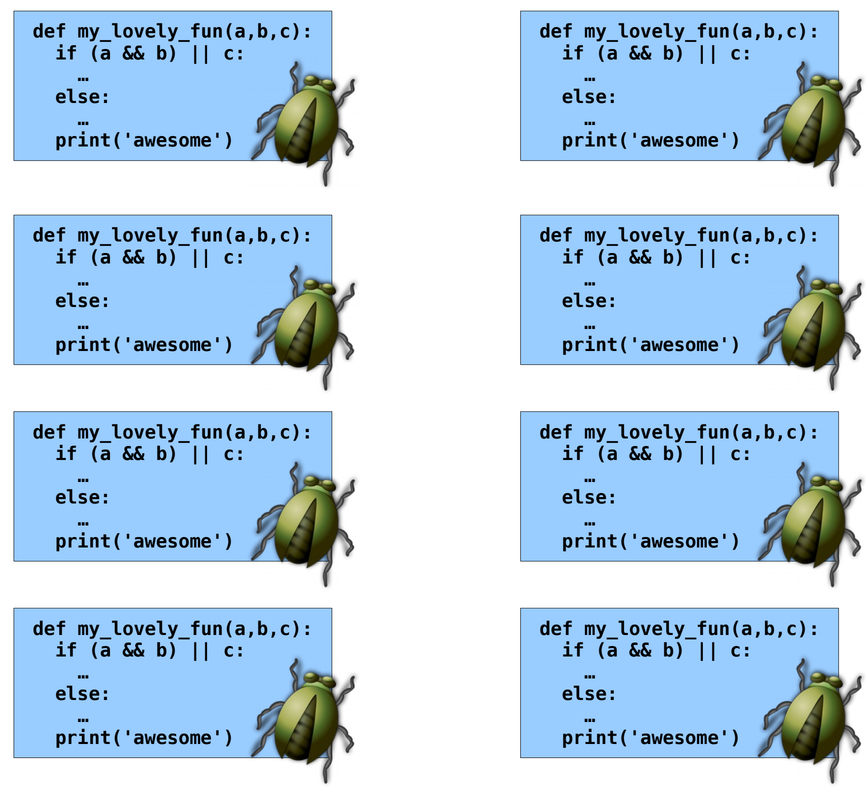
```
def my_lovely_fun(a,b,c):
   if (a & b) | c:
      …
   else:
      …
   print('awesome')
```

More common in safety critical systems where full coverage may be required.

Test Suite Adequacy

- Questions
  - Is a test suite good eno
  - What parts of software
- Metrics
  - Statement Coverage
  - Branch Coverage
  - MC/DC Coverage
  - Mutation Coverage

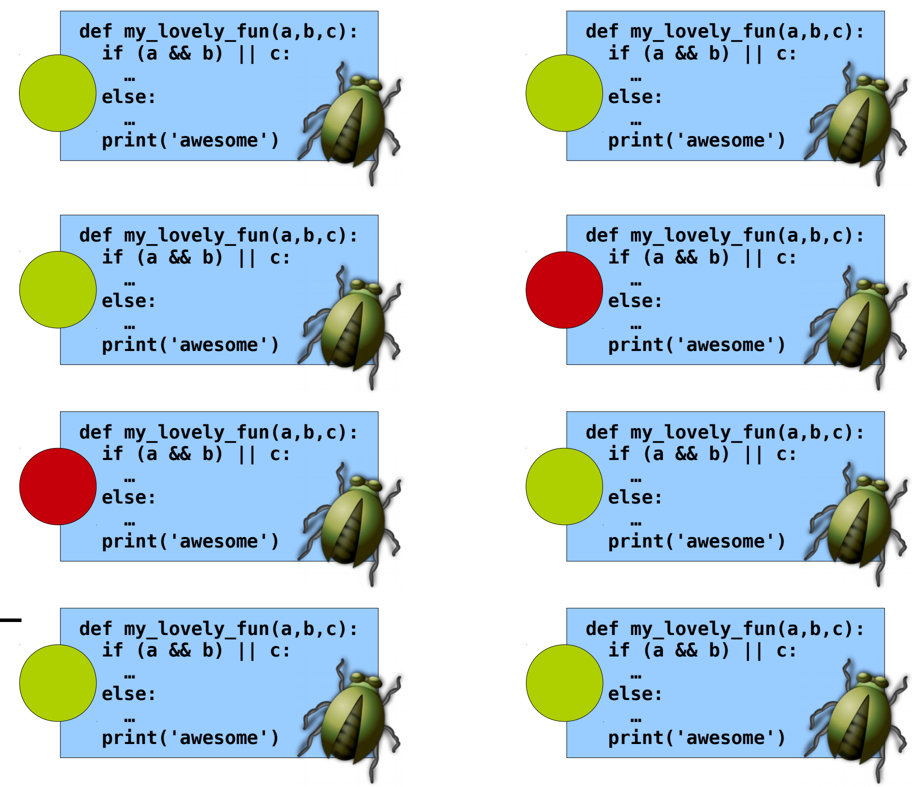
```
def my_lovely_fun(a,b,c):
  if (a & b) | c:
    …
  else:
    …
  print('awesome')
```



Test Suite Adequacy

- Questions
  - Is a test suite good eno
  - What parts of software
- Metrics
  - Statement Coverage
  - Branch Coverage
  - MC/DC Coverage
  - Mutation Coverage

```
def my_lovely_fun(a,b,c):
  if (a & b) | c:
    …
  else:
    …
  print('awesome')
```

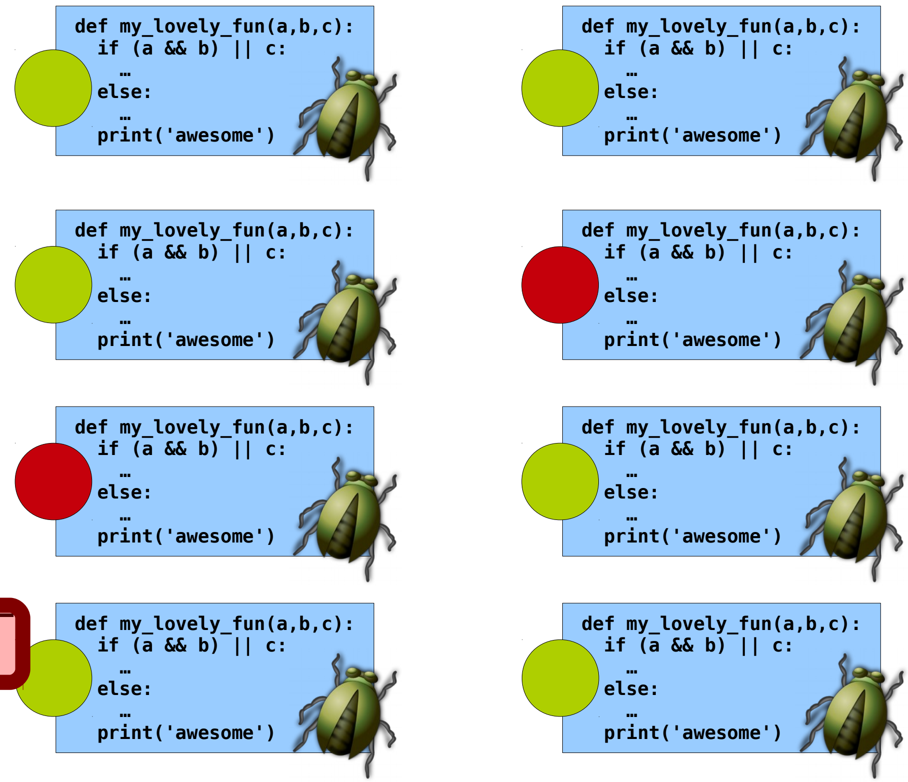


$$\text{score} = \frac{\text{\# covered/killed}}{\text{\# non-equivalent mutants}}$$

Test Suite Adequacy

- Questions
  - Is a test suite good eno
  - What parts of software
- Metrics
  - Statement Coverage
  - Branch Coverage
  - MC/DC Coverage
  - Mutation Coverage

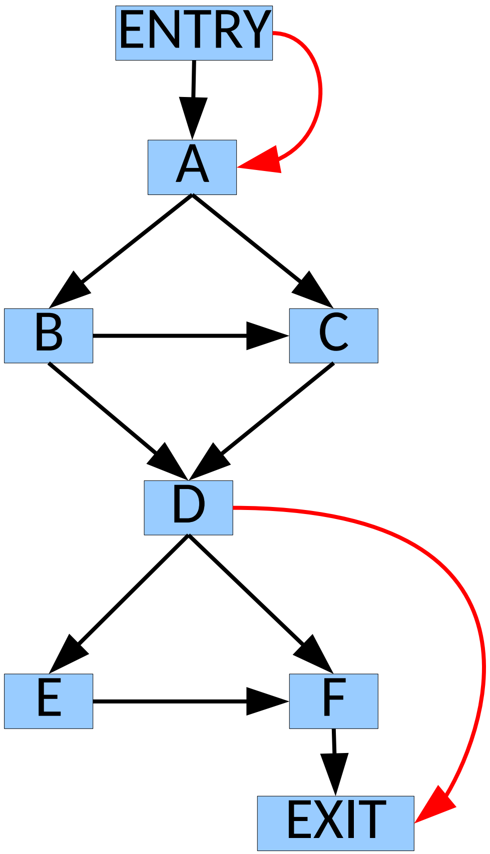
```
def my_lovely_fun(a,b,c):
  if (a & b) | c:
    …
  else:
    …
  print('awesome')
```



score = $\frac{\text{\# covered/killed}}{\text{\# non-equivalent mutants}}$

Test Suite Adequacy

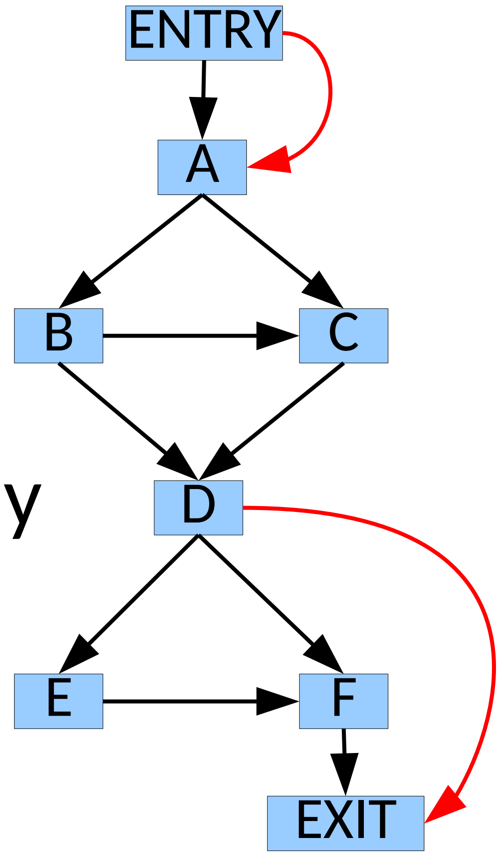
- Questions
  - Is a test suite good enough?
  - What parts of software need to be tested better?
- Metrics
  - Statement Coverage
  - Branch Coverage
  - MC/DC Coverage
  - Mutation Coverage
  - Path Coverage



Test Suite Adequacy

- Questions
  - Is a test suite good enough?
  - What parts of software need to be tested better?
- Metrics
  - Statement Coverage
  - Branch Coverage
  - MC/DC Coverage
  - Mutation Coverage
  - Path Coverage

Can apply EPP!



Test Suite Adequacy

- Questions
  - Is a test suite good enough?
  - What parts of software need to be tested better?
- Metrics
  - Statement Coverage
  - Branch Coverage
  - MC/DC Coverage
  - Mutation Coverage
  - Path Coverage
  - …

Test Suite Adequacy

- Questions
  - Is a test suite good enough?
  - What parts of software need to be tested better?
- Metrics
  - Statement Coverage
  - Branch Coverage
  - MC/DC Coverage
  - Mutation Coverage
  - Path Coverage

BUT reducing test suites for St, Br, MC/DC coverage decrease defect detection!

Generating Inputs

- Sample all possible inputs

```
for test in allPossibleInputs:
    check_test(test)
```

Generating Inputs

- Sample all possible inputs

```
for test in allPossibleInputs:
    check_test(test)
```

- Target specific goals

```
for s in statements:
    test = findTestThrough(s)
    check_test(test)
```

or other coverage criteria

Generating Inputs

- Sample all possible inputs

```
for test in allPossibleInputs:
    check_test(test)
```

- Target specific goals

```
for s in statements:
    test = findTestThrough(s)
    check_test(test)
```

or other coverage criteria

```
for i in inputModel:
    test = findRepresentative(i)
    check_test(test)
```

Generating Inputs

- Usually broken into black box & white box approaches

Generating Inputs

- Usually broken into black box & white box approaches
  - Black Box – Treat the program as opaque / unknown

    e.g. specification based, naive fuzzing,
    boundary value analysis, ...

Generating Inputs

- Usually broken into black box & white box approaches
  - Black Box – Treat the program as opaque / unknown
  - White Box – Program structure & semantics can be used

e.g. symbolic execution, call chain synthesis, white box fuzzing, boundary value analysis, ...

Generating Oracles

?

Generating Oracles

- Likely invariants?
- Careful variable selection & monitoring?
- Differential Testing
- Metamorphic Testing

A very open (hard) problem.

Interesting Problems

- Random Testing
- Test Suite Adequacy
- Test Suite Minimization
- Test Generation
- Oracle Generation
- Test Maintenance
- Performance Testing
- Field Testing
- Power Testing
- Test Prioritization
- …

Interesting Problems

- Random Testing
- Test Suite Adequacy
- Test Suite Minimization
- Test Generation
- Oracle Generation
- Test Maintenance
- Performance Testing
- Field Testing
- Power Testing
- Test Prioritization
- …

Test generation techniques have also proven to be critical in security research.

Fuzz Testing

- An approach for generating test inputs

Fuzz Testing

- An approach for generating test inputs
- Originally just feeding large random inputs to programs [Miller 1990]

```
./grep "02d6…" RandomFile
```

Fuzz Testing

- An approach for generating test inputs
- Originally just feeding large random inputs to programs [Miller 1990]

```
./grep “02d6…” RandomFile
```

It was distressingly effective at finding buffer overflows (25%-33% of programs).

Fuzz Testing

- An approach for generating test inputs
- Originally just feeding large random inputs to programs [Miller 1990]
- Now 2 main types

Fuzz Testing

- An approach for generating test inputs
- Originally just feeding large random inputs to programs [Miller 1990]
- Now 2 main types
  1) *Generational* (model based)
     - Creates entirely new inputs

Fuzz Testing

- An approach for generating test inputs
- Originally just feeding large random inputs to programs [Miller 1990]
- Now 2 main types

  1) Generational (model based)
    - Creates entirely new inputs
    - Needs a *model* for the input

Fuzz Testing

- An approach for generating test inputs
- Originally just feeding large random inputs to programs [Miller 1990]
- Now 2 main types

  1) Generational (model based)

  - Creates entirely new inputs
  - Needs a *model* for the input

  a*bc(d|e)c*

Fuzz Testing

- An approach for generating test inputs
- Originally just feeding large random inputs to programs [Miller 1990]
- Now 2 main types
  1) *Generational* (model based)
    - Creates entirely new inputs
    - Needs a *model* for the input

a*bc(d|e)c*

$A \rightarrow aAb$
$A \rightarrow cA$
$A \rightarrow \varepsilon$

Fuzz Testing

- An approach for generating test inputs
- Originally just feeding large random inputs to programs [Miller 1990]
- Now 2 main types

  1) Generational (model based)

  - Creates entirely new inputs
  - Needs a *model* for the input

a*bc(d|e)c*

...

$A \rightarrow aAb$
$A \rightarrow cA$
$A \rightarrow \varepsilon$

Fuzz Testing

- An approach for generating test inputs
- Originally just feeding large random inputs to programs [Miller 1990]
- Now 2 main types
  1) *Generational* (model based)
  2) *Mutational* (heuristic change based)
     - Modify an existing test suite

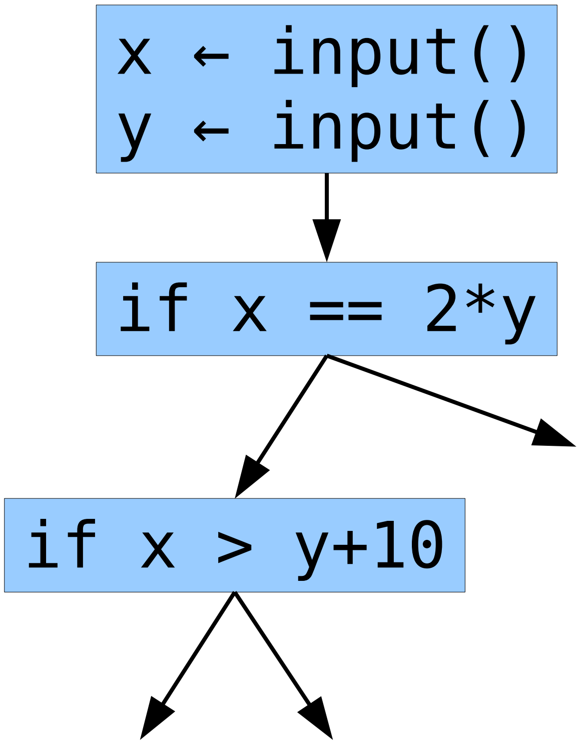
Fuzz Testing

- An approach for generating test inputs
- Originally just feeding large random inputs to programs [Miller 1990]
- Now 2 main types
  1) *Generational* (model based)
  2) *Mutational* (heuristic change based)
     - Modify an existing test suite
     - Seeing a resurgance via *AFL* & *libFuzzer*

Symbolic Execution

Cadar & Sen, 2013

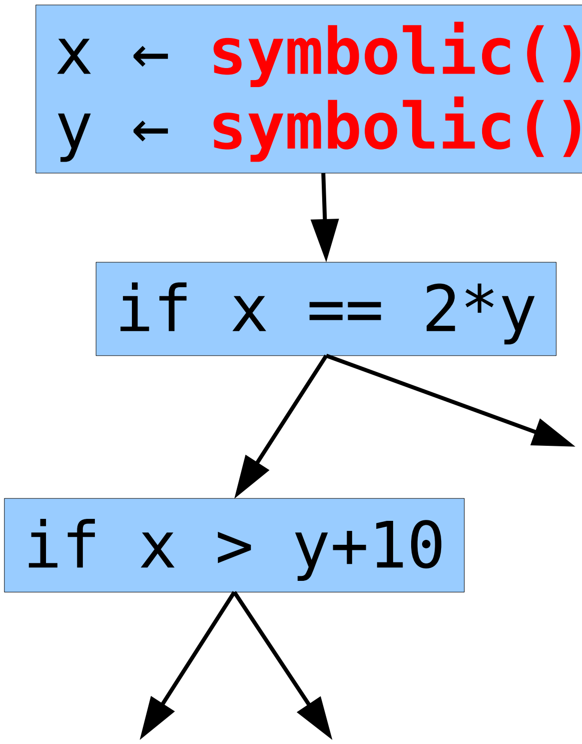
- An approach for generating test inputs.



Symbolic Execution

Cadar & Sen, 2013

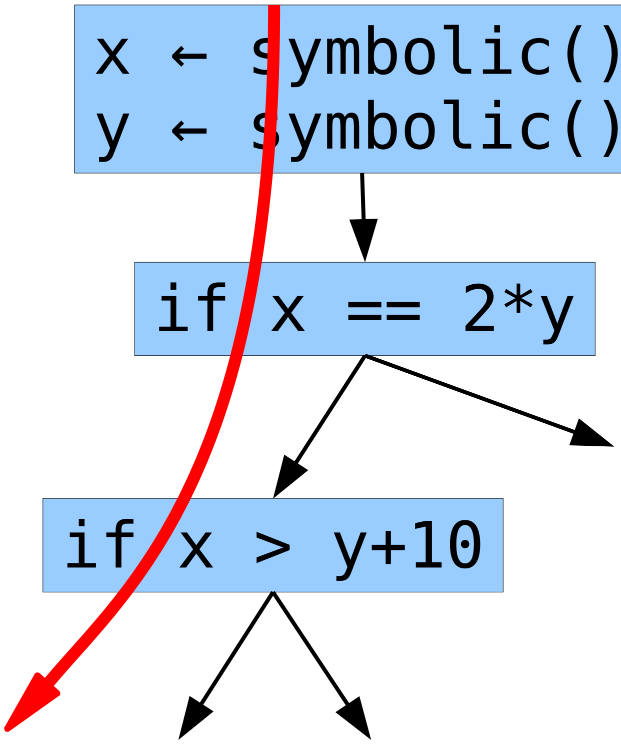
- An approach for generating test inputs.
- Replace the concrete inputs of a program with symbolic values



Symbolic Execution

Cadar & Sen, 2013

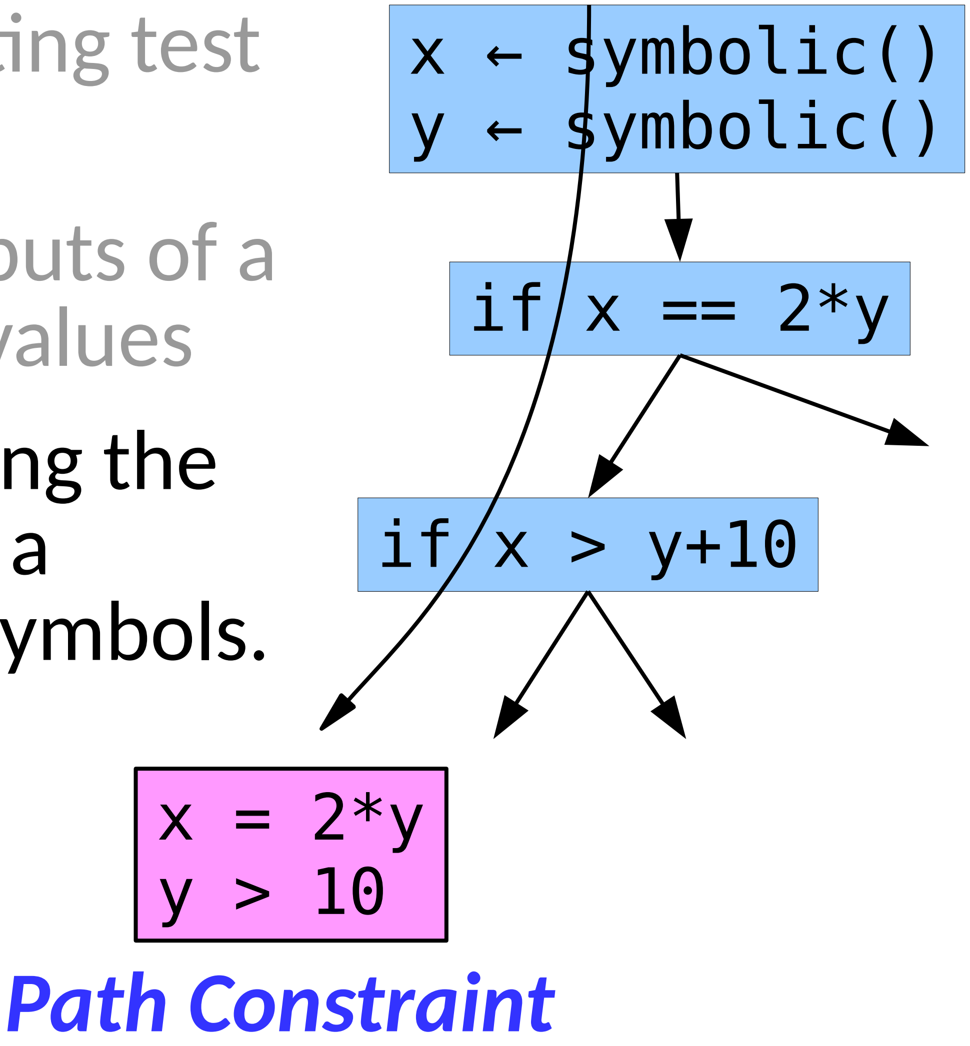
- An approach for generating test inputs.
- Replace the concrete inputs of a program with symbolic values
- Execute along a path using the symbolic values to build a formula over the input symbols.



Symbolic Execution

Cadar & Sen, 2013

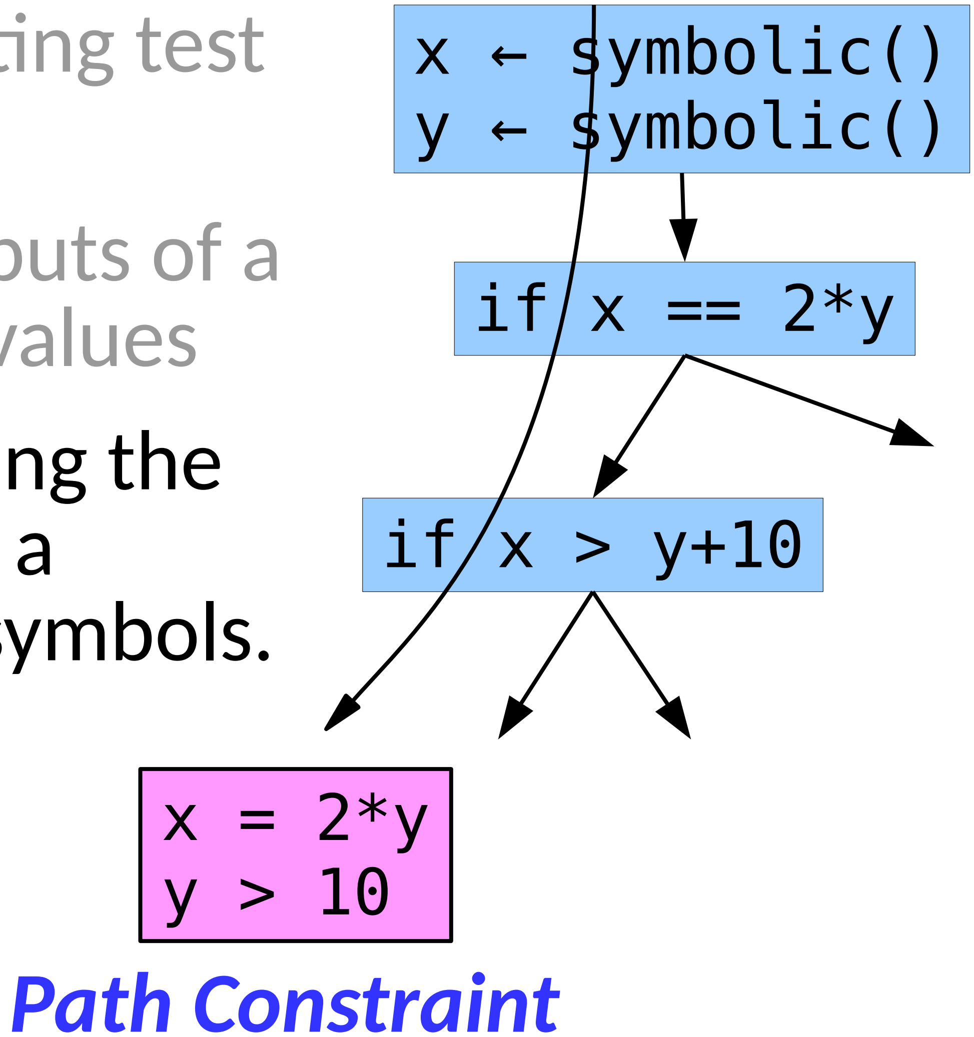
- An approach for generating test inputs.
- Replace the concrete inputs of a program with symbolic values
- Execute along a path using the symbolic values to build a formula over the input symbols.



Symbolic Execution

Cadar & Sen, 2013

- An approach for generating test inputs.
- Replace the concrete inputs of a program with symbolic values
- Execute along a path using the symbolic values to build a formula over the input symbols.

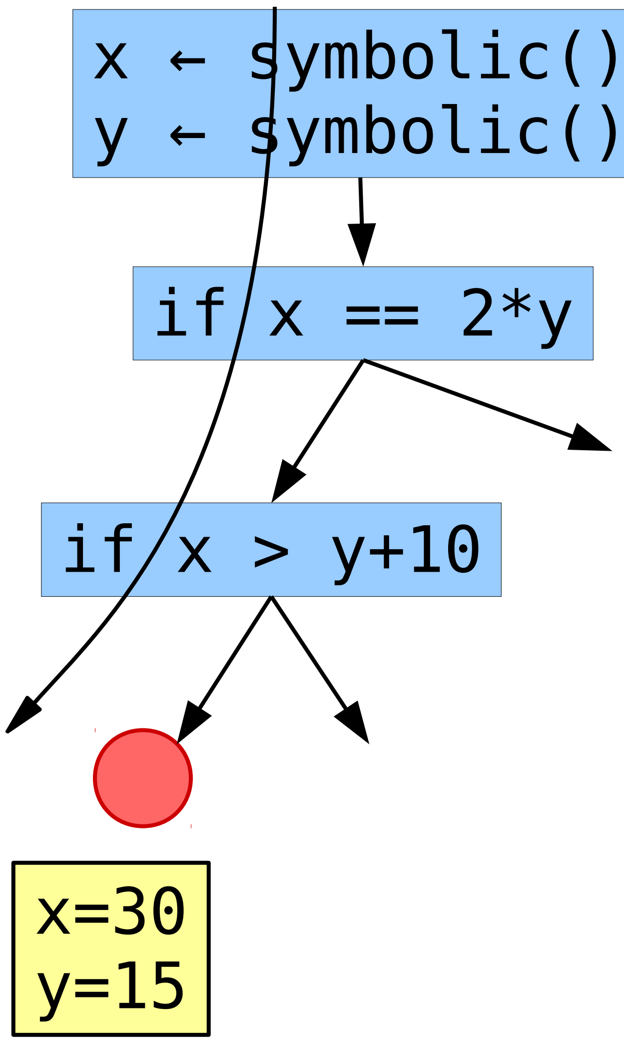


A path constraint represents all executions along that path

Symbolic Execution

Cadar & Sen, 2013

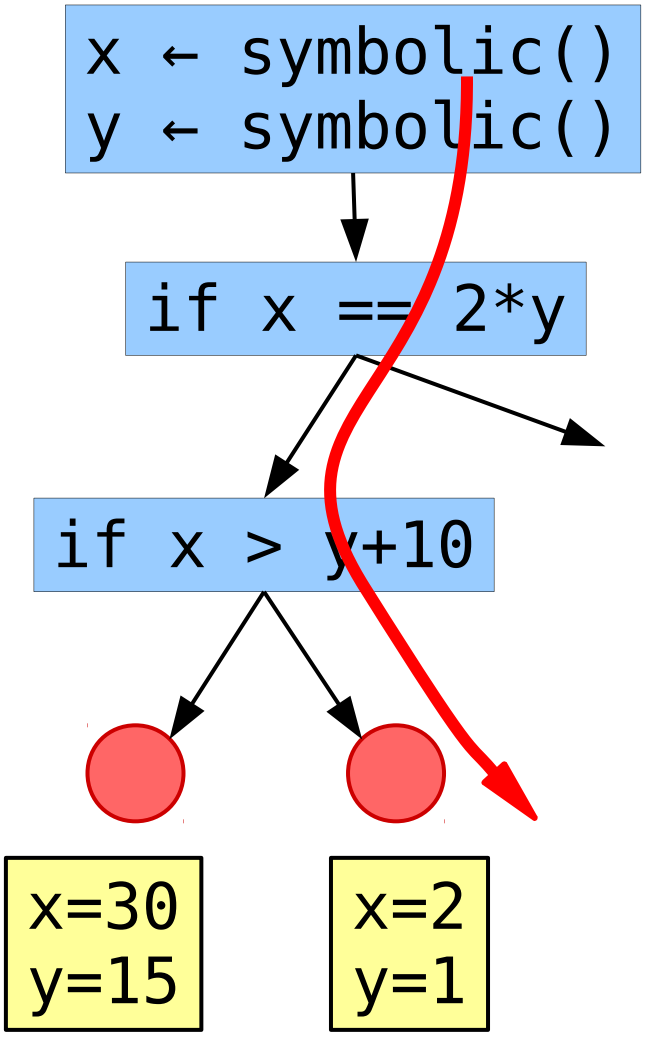
- An approach for generating test inputs.
- Replace the concrete inputs of a program with symbolic values
- Execute along a path using the symbolic values to build a formula over the input symbols.
- Solve for the symbolic symbols to find inputs that yield the path.



Symbolic Execution

- An approach for generating test inputs.
- Replace the concrete inputs of a program with symbolic values
- Execute along a path using the symbolic values to build a formula over the input symbols.
- Solve for the symbolic symbols to find inputs that yield the path.

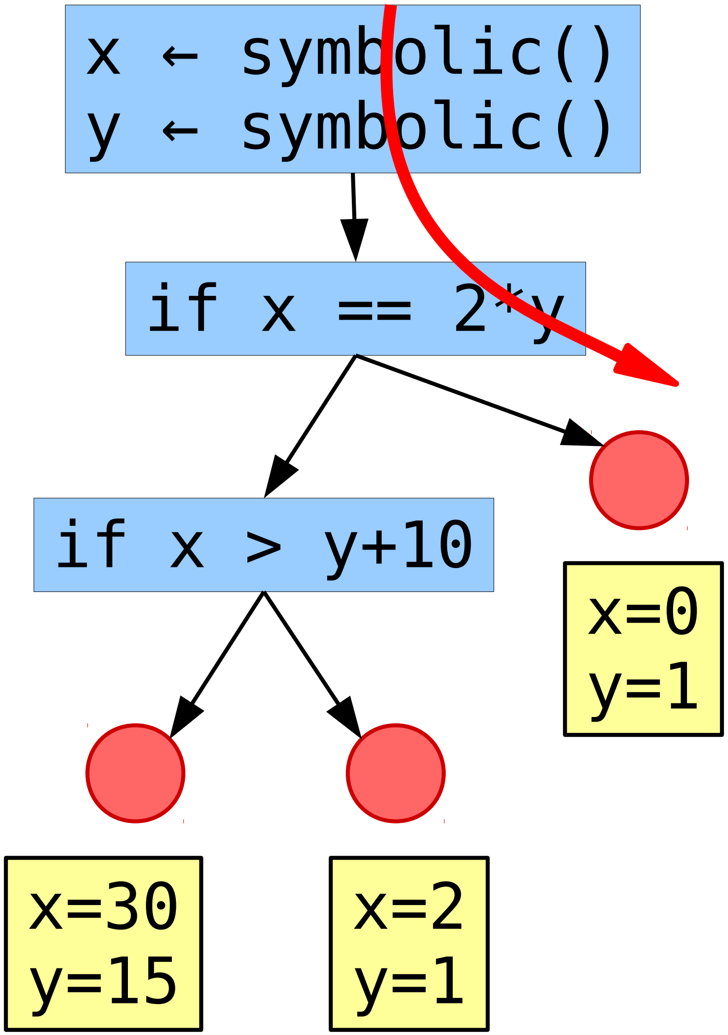
Cadar & Sen, 2013



Symbolic Execution

Cadar & Sen, 2013

- An approach for generating test inputs.
- Replace the concrete inputs of a program with symbolic values
- Execute along a path using the symbolic values to build a formula over the input symbols.
- Solve for the symbolic symbols to find inputs that yield the path.



How Can We Solve Constraints?

- SMT Solvers
  - Satisfiability Modulo Theories
  - SAT with extra logic
  - Standard interfaces through SMTLIB2

How Can We Solve Constraints?

- SMT Solvers
  - Satisfiability Modulo Theories
  - SAT with extra logic
  - Standard interfaces through SMTLIB2

```
x = 2*y
y > 10
```

```
(declare-const x Int)
(declare-const y Int)
(assert (= x (* 2 y)))
(assert (> y 10))
(check-sat)
(get-model)
```

How Can We Solve Constraints?

- SMT Solvers
  - Satisfiability Modulo Theories
  - SAT with extra logic
  - Standard interfaces through SMTLIB2

```
x = 2*y
y > 10
```

```
(declare-const x Int)
(declare-const y Int)
(assert (= x (* 2 y)))
(assert (> y 10))
(check-sat)
(get-model)
```

Z3 →

How Can We Solve Constraints?

- SMT Solvers
  - Satisfiability Modulo Theories
  - SAT with extra logic
  - Standard interfaces through SMTLIB2

```
x = 2*y
y > 10
```

```
(declare-const x Int)
(declare-const y Int)
(assert (= x (* 2 y)))
(assert (> y 10))
(check-sat)
(get-model)
```

Z3 →

```
x=22
y=11
```

```
sat
(model
   (define-fun y () Int 11)
   (define-fun x () Int 22)
)
```

How Can We Solve Constraints?

- SMT Solvers
  - Satisfiability Modulo Theories
  - SAT with extra logic
  - Standard interfaces through SMTLIB2

```
x = 2*y
y > 10
```

```
(declare-const x Int)
(declare-const y Int)
(assert (= x (* 2 y)))
(assert (> y 10))
(check-sat)
(get-model)
```

Z3 →

```
x=22
y=11
```

```
sat
(model
   (define-fun y () Int 11)
   (define-fun x () Int 22)
)
```

Try it online:
http://www.rise4fun.com/Z3/tutorial/

Useful Questions

- If φ holds after a statement,
  what must have been true at the point before?
  - ***weakest precondition***

Useful Questions

- If φ holds after a statement,
  what must have been true at the point before?
  - ***weakest precondition***
- If φ holds before a statement,
  what can we guarantee to be true after?
  - ***strongest poscondition***

Useful Questions

- If φ holds after a statement, what must have been true at the point before?
  - ***weakest precondition***
- If φ holds before a statement, what can we guarantee to be true after?
  - ***strongest poscondition***

e.g. Given two versions of a program v_1, v_2
and assertions on output φ_i in each from an input I

What is $wp(\varphi_1) \land \neg wp(\varphi_2)$?

Useful Questions

- If φ holds after a statement,
  what must have been true at the before?
  - ***weakest precondition***
- If φ holds before a statement,
  what can we guarantee to be true after?
  - ***strongest poscondition***

e.g. Given two versions of a program v_1,v_2
and assertions on output φ_i in each from an input I

What is $wp(\varphi_1) \wedge \neg wp(\varphi_2)$? $wp(\varphi_1) \Rightarrow wp(\varphi_2)$

Useful Questions

- If φ holds after a statement,
  what must have been true at the before?
  - ***weakest precondition***
- If φ holds before a statement,
  what can we guarantee to be true after?
  - ***strongest poscondition***

e.g. Given two versions of a program v_1,v_2
and assertions on output φ_i in each from an input I

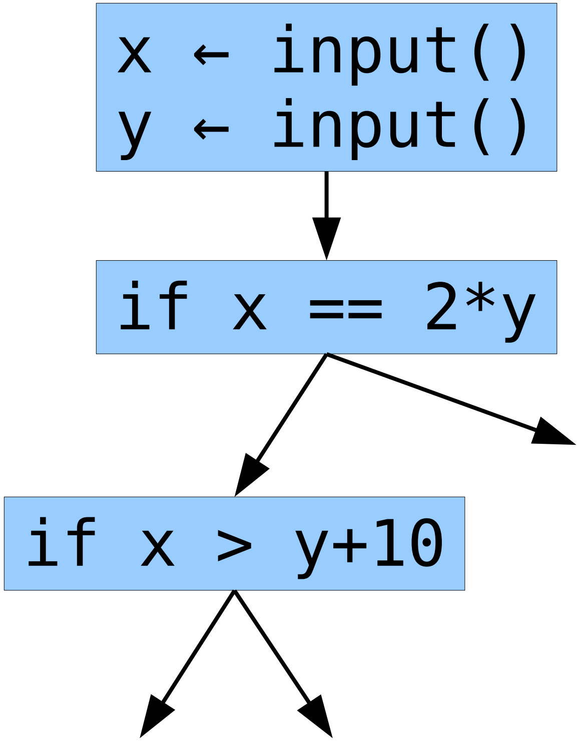
What is $wp(\varphi_1) \wedge \neg wp(\varphi_2)$? $wp(\varphi_1) \Rightarrow wp(\varphi_2)$

What is the intuitive meaning?

Exploring the Execution Tree

- The possible paths of a program form an ***execution tree***.

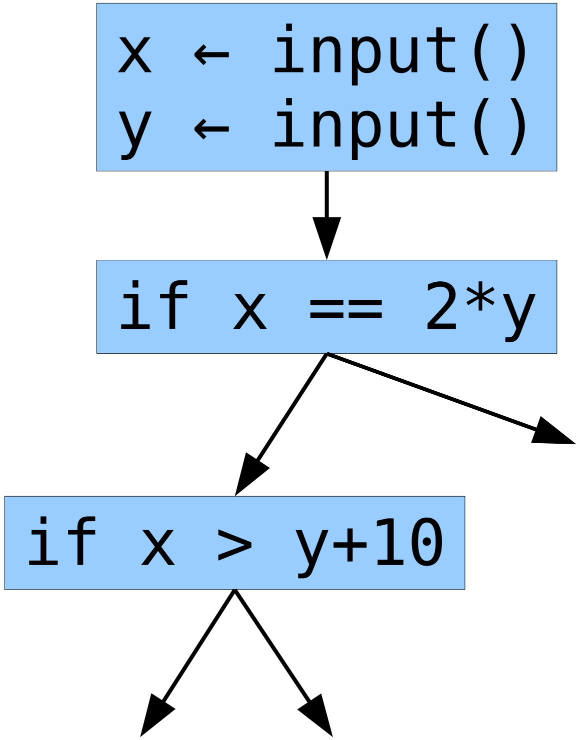
Cadar & Sen, 2013



Exploring the Execution Tree

- The possible paths of a program form an ***execution tree***.
- Traversing the tree will yield tests for all paths.

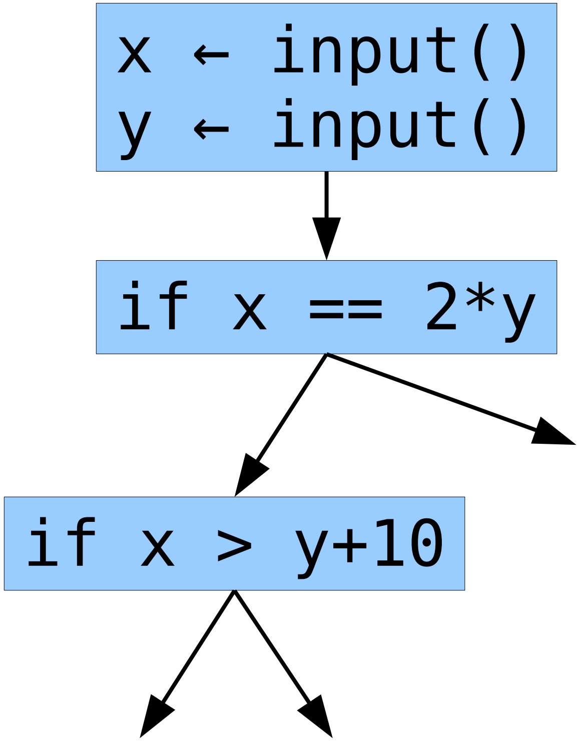
Cadar & Sen, 2013



Exploring the Execution Tree

- The possible paths of a program form an ***execution tree***.
- Traversing the tree will yield tests for all paths.
- Mechanizing the traversal yields two main approaches

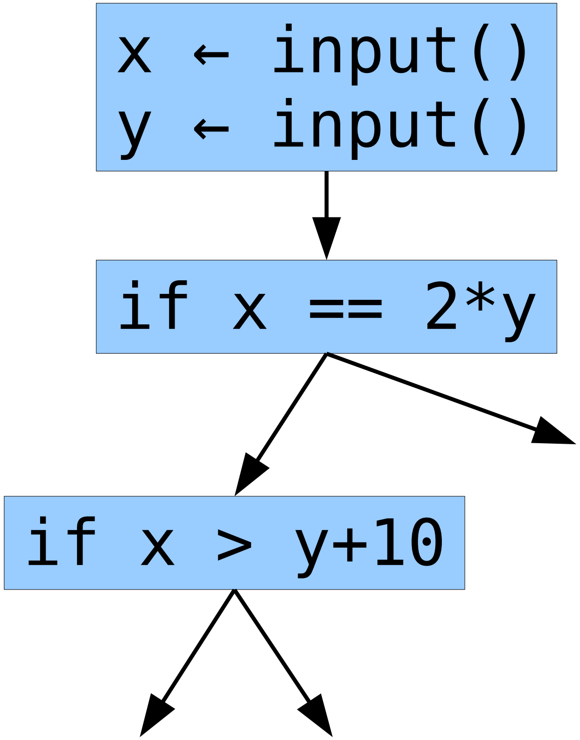
Cadar & Sen, 2013



Exploring the Execution Tree

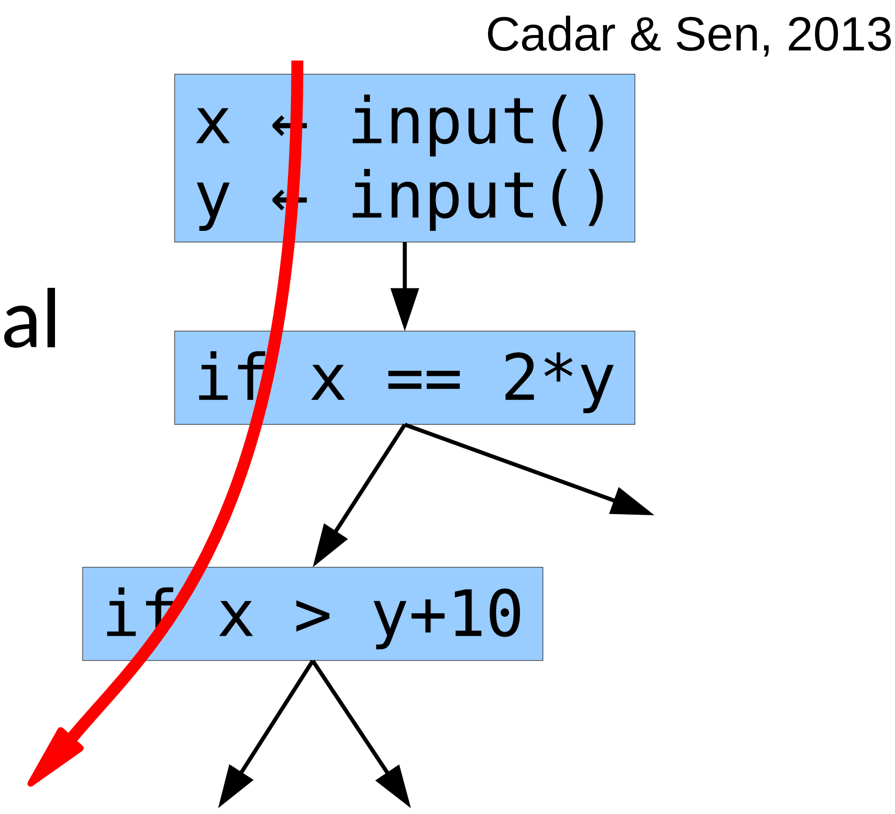
- The possible paths of a program form an ***execution tree***.
- Traversing the tree will yield tests for all paths.
- Mechanizing the traversal yields two main approaches
  - Concolic (dynamic symbolic)

Cadar & Sen, 2013



Exploring the Execution Tree

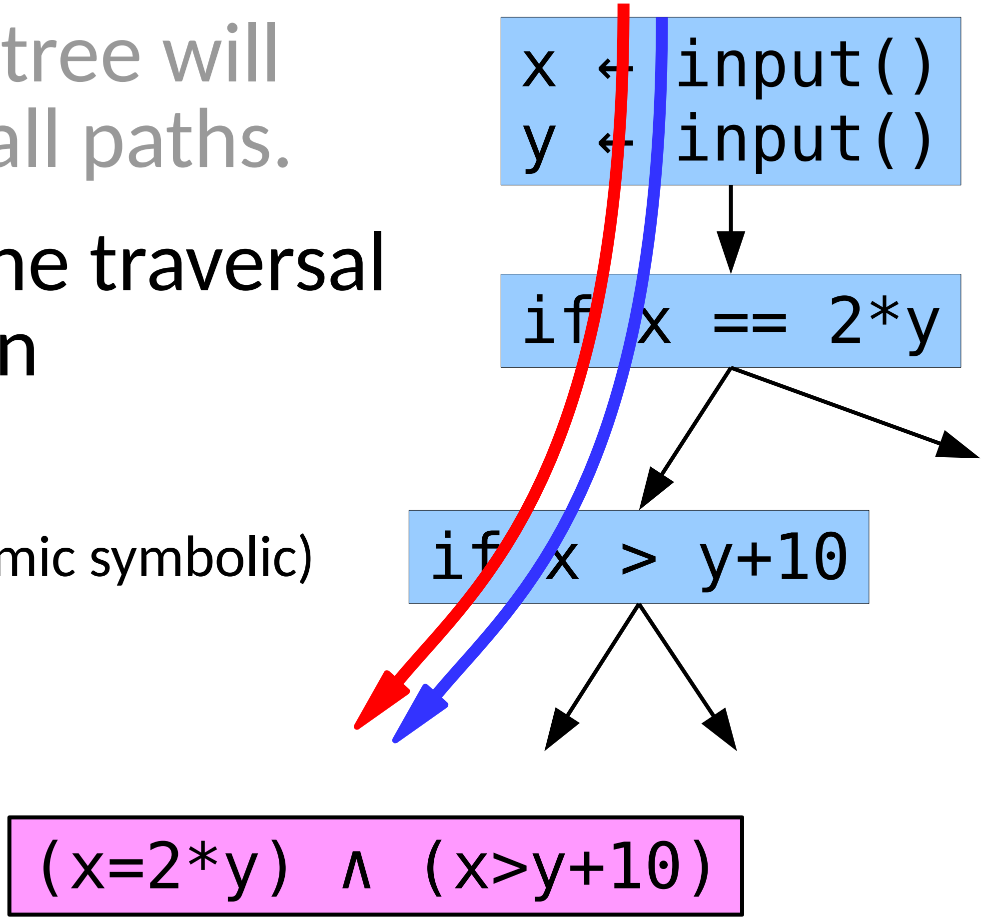
- The possible paths of a program form an ***execution tree***.
- Traversing the tree will yield tests for all paths.
- Mechanizing the traversal yields two main approaches
  - Concolic (dynamic symbolic)



Exploring the Execution Tree

- The possible paths of a program form an ***execution tree***.
- Traversing the tree will yield tests for all paths.
- Mechanizing the traversal yields two main approaches
  - Concolic (dynamic symbolic)

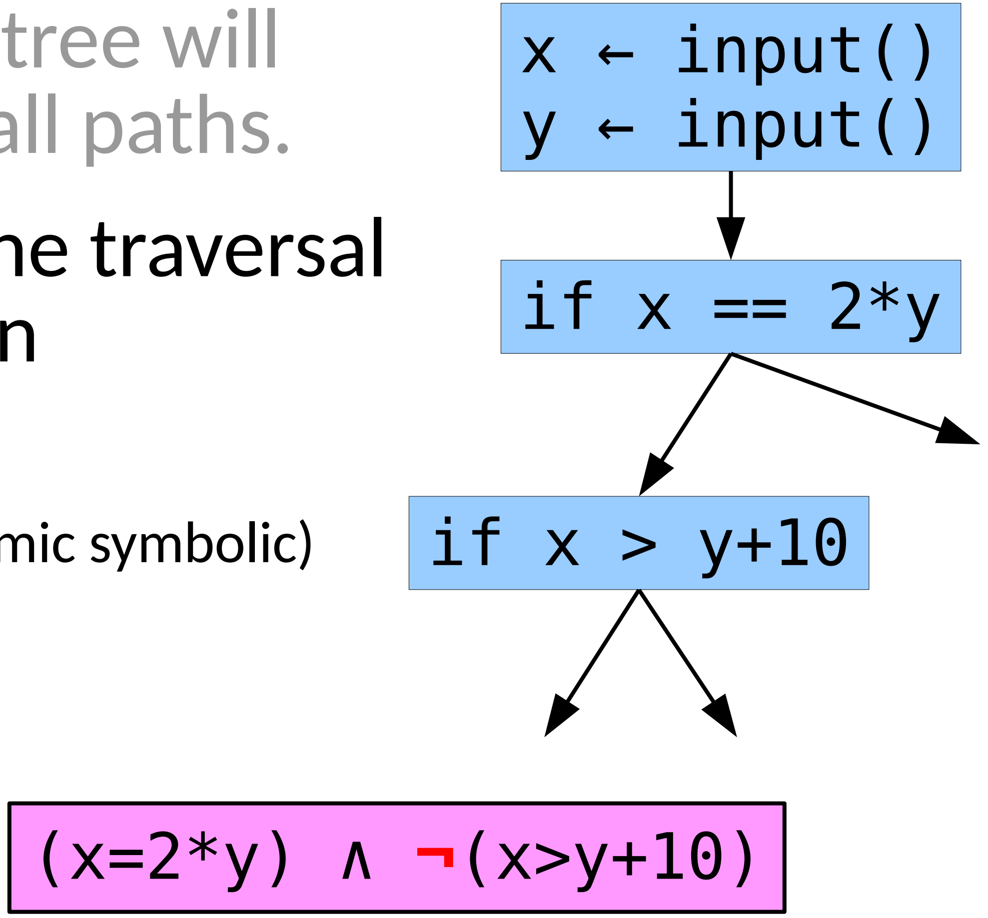
Cadar & Sen, 2013



Exploring the Execution Tree

- The possible paths of a program form an ***execution tree***.
- Traversing the tree will yield tests for all paths.
- Mechanizing the traversal yields two main approaches
  - Concolic (dynamic symbolic)

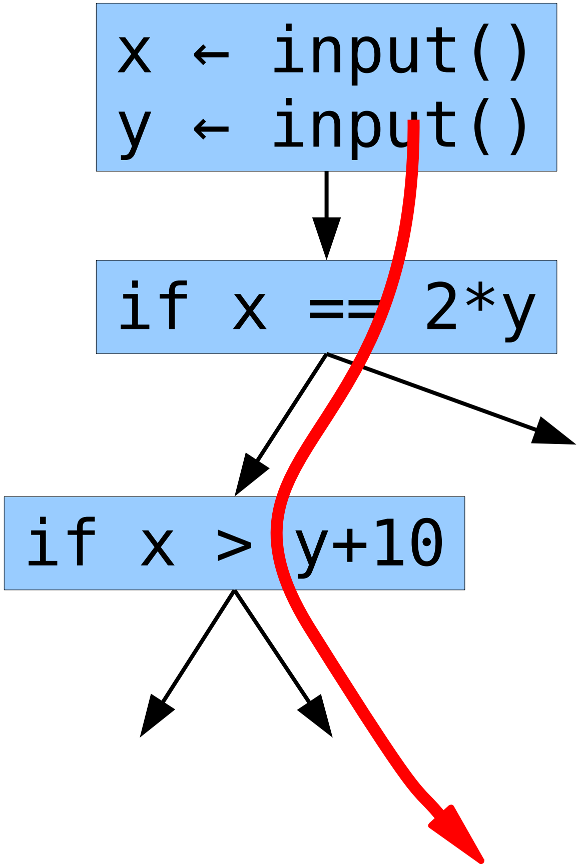
Cadar & Sen, 2013



Exploring the Execution Tree

- The possible paths of a program form an ***execution tree***.
- Traversing the tree will yield tests for all paths.
- Mechanizing the traversal yields two main approaches
  - Concolic (dynamic symbolic)

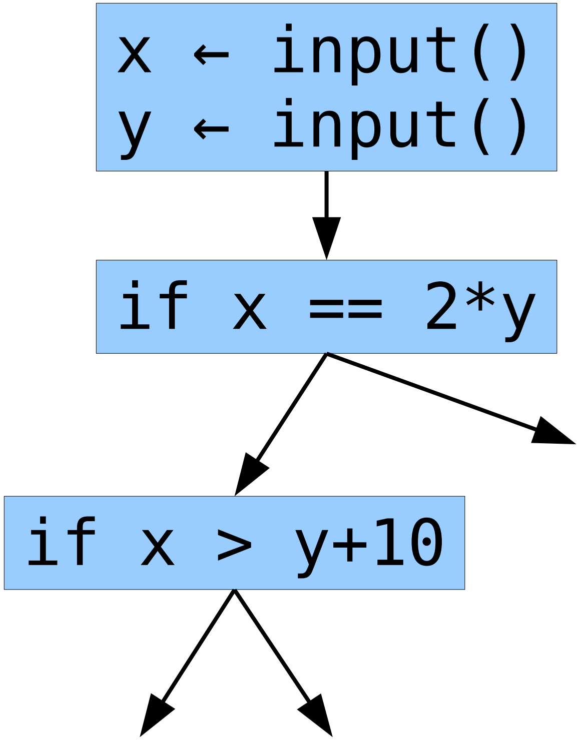
Cadar & Sen, 2013



Exploring the Execution Tree

- The possible paths of a program form an ***execution tree***.
- Traversing the tree will yield tests for all paths.
- Mechanizing the traversal yields two main approaches
  - Concolic (dynamic symbolic)
  - Execution Generated Testing

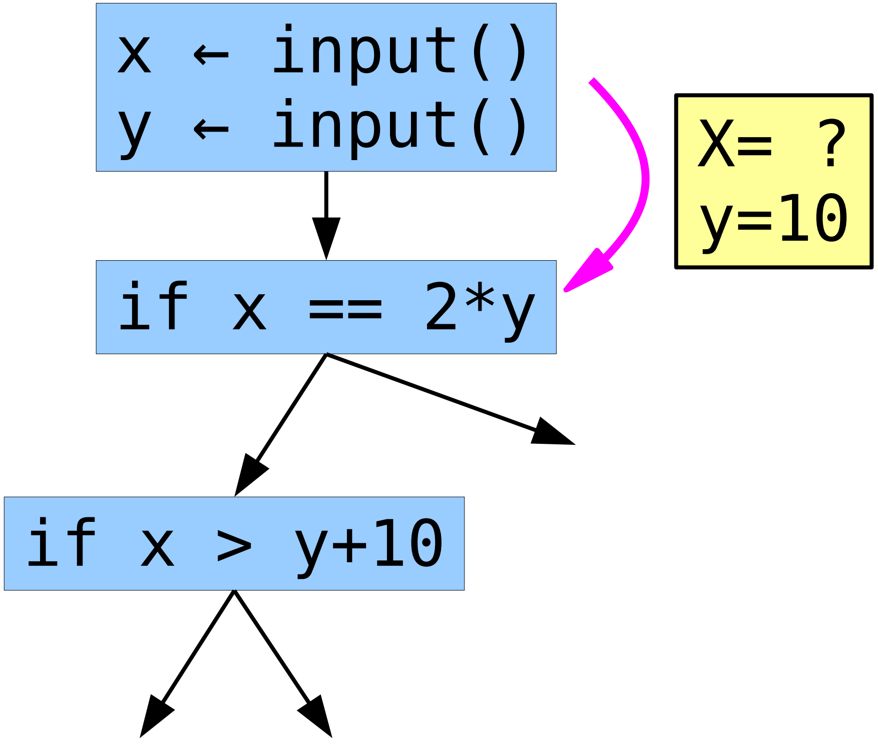
Cadar & Sen, 2013



Exploring the Execution Tree

- The possible paths of a program form an ***execution tree***.
- Traversing the tree will yield tests for all paths.
- Mechanizing the traversal yields two main approaches
  - Concolic (dynamic symbolic)
  - Execution Generated Testing

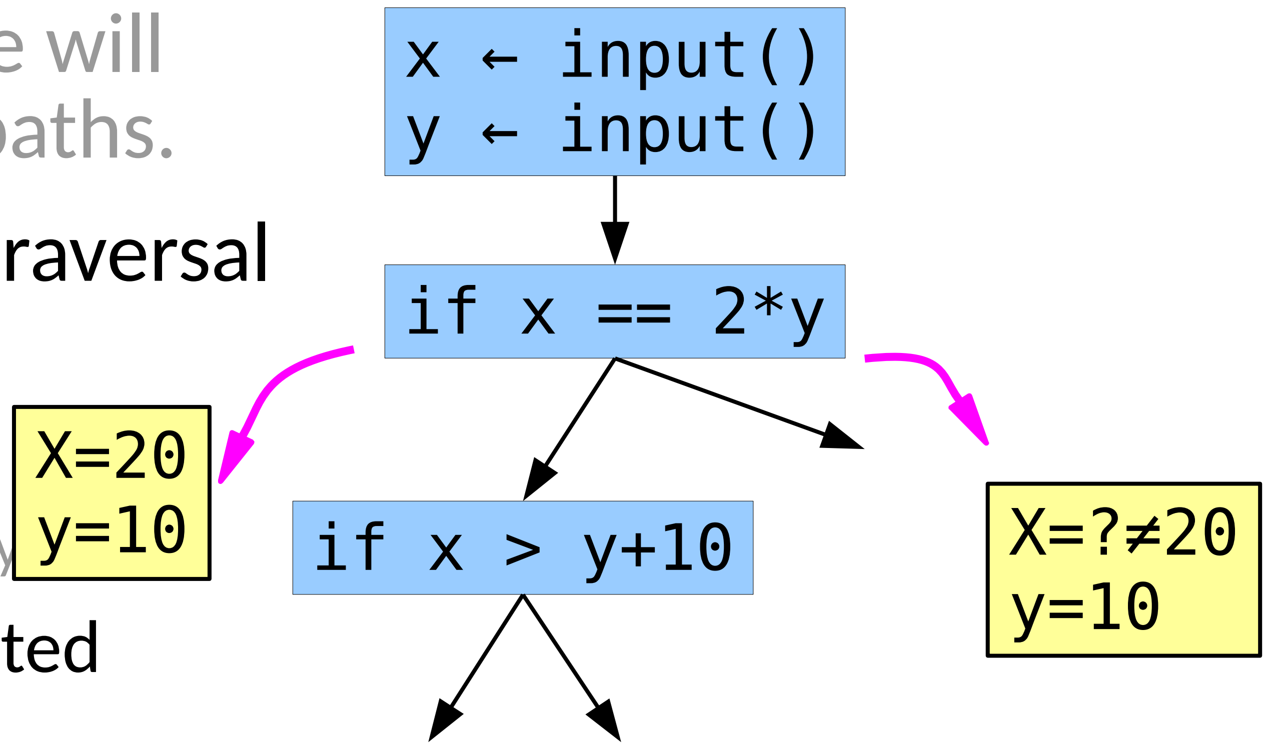
Cadar & Sen, 2013



Exploring the Execution Tree

- The possible paths of a program form an ***execution tree***.
- Traversing the tree will yield tests for all paths.
- Mechanizing the traversal yields two main approaches
  - Concolic (dynamic sy
  - Execution Generated Testing

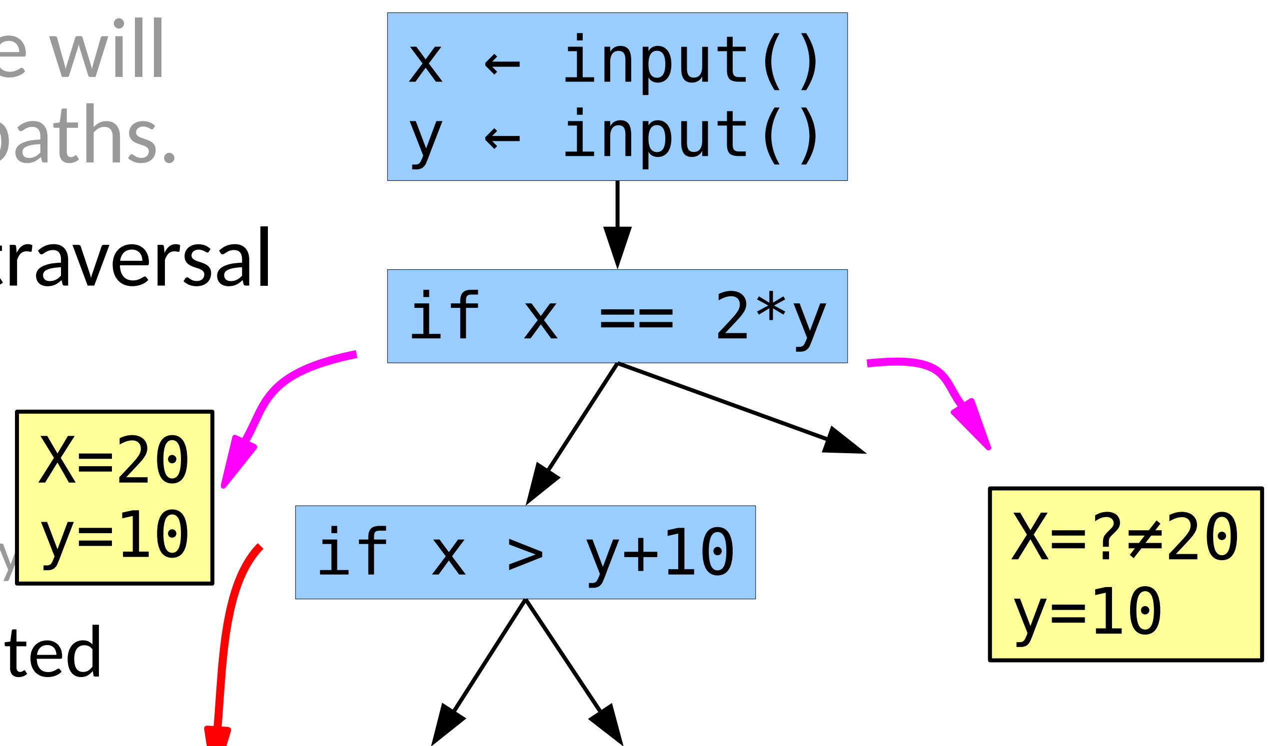
Cadar & Sen, 2013



Exploring the Execution Tree

- The possible paths of a program form an ***execution tree***.
- Traversing the tree will yield tests for all paths.
- Mechanizing the traversal yields two main approaches
  - Concolic (dynamic sy
  - Execution Generated Testing

Cadar & Sen, 2013



Execution on this side is concrete from this point on.

Symbolic Execution

- Increasingly scalable every year

Symbolic Execution

- Increasingly scalable every year
- Can automatically generate test inputs from constraints

Symbolic Execution

- Increasingly scalable every year
- Can automatically generate test inputs from constraints
- The resulting symbolic formulae have many uses beyond just testing.

Symbolic Execution

- Increasingly scalable every year
- Can automatically generate test inputs from constraints
- The resulting symbolic formulae have many uses beyond just testing.

Try it out:
1) https://github.com/klee/klee
2) Symbolic PathFinder
3) http://research.microsoft.com/Pex/
4) http://angr.io/