Testing

AZADEH FARZAN

FALL 2017
Testing

- Developers cannot prevent/eliminate all defects during development.

- Software must be tested **before delivery** to the users.

- The responsibility of the tester is to design tests that:
  - reveal **defects**.
  - can be used to evaluate **performance, usability, and reliability** of software.
Why is Random Testing not good enough?

```c
1. void f(int x, int y) {
2.   int z = 2*y;
3.   if (x == 100000) {
4.     if (x < z) {
5.       assert(0); /* error */
6.     }
7.   }
8. }
```
Why is Random Testing not good enough?

Random testing would require an expected large number of tests to discover the error.
Let’s discuss methods beyond random testing ...
Two Major Forms of Testing

- **Black Box Testing**
  - Functional or specification test strategies
  - No knowledge of the inner structure of the software.
  - The tester has only the knowledge of what the software does (specification).

- **White Box Testing**
  - Focuses on the inner structure of software.
  - The tester has knowledge of the structure in form of some model of the program.
Black Box Testing
Black Box Testing

- The size of the software under test may vary.
Black Box Testing

- The size of the software under test may vary.
- The description (functionality) of the software may come from formal specification, or other sources (Pre/Post-conditions, ...).
Black Box Testing

- The **size** of the software under test may vary.

- The **description** (functionality) of the software may come from **formal specification**, or other sources (Pre/Post-conditions, ...).

- The tester specifies **inputs and the expected outputs** and runs the software.
Black Box Testing

- The size of the software under test may vary.
- The description (functionality) of the software may come from formal specification, or other sources (Pre/Post-conditions, ...).
- The tester specifies inputs and the expected outputs and runs the software.
- Specially useful for revealing requirement and specification defects.
Black Box Strategies
Black Box Strategies

- Naive: all possibilities
Black Box Strategies

- **Naive**: all possibilities
- **Random Testing**: Opinions divided on its effectiveness.
Black Box Strategies

- **Naive**: all possibilities
- **Random Testing**
  - Opinions divided on its effectiveness.
- **Equivalence Class Partitioning**
  - Software specification usually derives interesting input conditions.
Black Box Strategies

- **Naive**: all possibilities
- **Random Testing**
  - Opinions divided on its effectiveness.
- **Equivalence Class Partitioning**
  - Software specification usually derives interesting input conditions.
- **Boundary Value Analysis**
  - Boundaries of equivalence classes are more likely to cause defects.
White Box Testing
White Box Testing

- More time-consuming: usually applied to smaller-sized software units.
White Box Testing

- More time-consuming: usually applied to smaller-sized software units.
- The tester uses a model (the code or a pseudocode-like representation) to select test cases that exercise specific internal structure elements.
White Box Testing

- More time-consuming: usually applied to smaller-sized software units.
- The tester uses a model (the code or a pseudocode-like representation) to select test cases that exercise specific internal structure elements.
- Specially useful for revealing logic and sequence defects, initialization defects, and data flow defects.
White Box Testing
Test Adequacy Criteria

- Minimal standards for testing
- Quantitative objectives for testing (a measure for when testing can be stopped).
- Selection of test data for a specific program property
- A criterion usually corresponds to a certain type of program structures.
  - statements, branches, conditions, paths, data flows, ...
Motivating Example

```c
void test(int x, int y) {
    if (x > 0) {
        if (y == hash(x))
            S0;
        else
            S1;
        if (x > 3 && y > 10)
            S3;
        else
            S4;
    }
}
```

```c
int hash(x) {
    if (0<=x<=10) return x*10;
    else return 0;
}
```

How do we obtain Statement Coverage?

Random Inputs might work if you are moderately lucky. But there is a better way! Win the Lottery.
Motivating Example

```c
void test(int x, int y) {
    if (x > 0) {
        if (y == hash(x))
            S1;
        else
            if (x > 3 && y > 10)
                S3;
            else
                S4;
    }
}

int hash(x) {
    if (0<=x<=10) return x*10;
    else return 0;
}
```

How do we obtain Statement Coverage?
Random Inputs might work if you are moderately lucky. But there is a better way! Win the Lottery.
Coverage
Coverage

- A test data set is statement adequate (also branch or condition) if it causes all the statements (or branches or conditions) to be executed.
Coverage

- A test data set is statement adequate (also branch or condition) if it causes all the statements (or branches or conditions) to be executed.

- These adequacy criteria and their requirements lead rise to the concept of coverage analysis.
Coverage

- A test data set is statement adequate (also branch or condition) if it causes all the statements (or branches or conditions) to be executed.

- These adequacy criteria and their requirements lead rise to the concept of coverage analysis.

- Coverage analysis is used to set testing goals and to develop test data sets.
Coverage

- A test data set is statement adequate (also branch or condition) if it causes all the statements (or branches or conditions) to be executed.

- These adequacy criteria and their requirements lead rise to the concept of coverage analysis.

- Coverage analysis is used to set testing goals and to develop test data sets.

- Testers usually refer to the adequacy criteria as coverage criteria.
A test data set is **statement adequate** (also branch or condition) if it causes all the statements (or branches or conditions) to be executed.

These adequacy criteria and their requirements lead rise to the concept of **coverage analysis**.

Coverage analysis is used to set **testing goals** and to **develop test data sets**.

Testers usually refer to the adequacy criteria as **coverage criteria**.

When full coverage cannot be achieved, the percentage of coverage is usually referred to as **degree of coverage**.
Example

Test cases for:

branch coverage:

1. pos_sum(a, num_of_entries, sum)
2. sum = 0
3. inti = 1
4. while (i <= num_of_entries)
5. if a[i] > 0
6.     sum = sum + a[i]
7.     endif
8.     i = i + 1
9. end while
10. end pos_sum

FIG. 5.2

Code sample with branch and loop.

so that each decision element in the code (if-then, case, loop) executes with all possible outcomes at least once. In terms of the control flow model, this requires that all the edges in the corresponding flow graph must be exercised at least once. Complete decision coverage is considered to be a stronger coverage goal than statement coverage since its satisfaction results in satisfying statement coverage as well (covering all the edges in a flow graph will ensure coverage of the nodes). In fact, the statement coverage goal is so weak that it is not considered to be very useful for revealing defects. For example, if the defect is a missing statement it may remain undetected by tests satisfying complete statement coverage. The reader should be aware that in spite of the weakness, even this minimal coverage goal is not required in many test plans.

Decision (branch) coverage for the code example in Figure 5.2, requires test cases to be developed for the two decision statements, that is, the four true/false edges in the control flow graph of Figure 5.3. Input values must ensure execution the true/false possibilities for the decisions in line 4 (while loop) and line 5 (if statement). Note that the "if" statement has a "null else" component, that is, there is no "else" part. However, we include a test that covers both the true and false conditions for the statement.

A possible test case that satisfies 100% decision coverage is shown in Table 5.1. The reader should note that the test satisfies both the branch
Example

Test cases for:

branch coverage:

\[ a = [1, -45, 3] \]
Example

Test cases for:

branch coverage:

\[ a=[1,-45,3] \]

1. pos_sum(a, num_of_entries, sum)
2. sum = 0
3. inti = 1
4. while (i <= num_of_entries)
5. if a[i] > 0
6. sum = sum + a[i]
   endif
7. i = i + 1
end while
8. end pos_sum

if(age <65 and married == true)
do X
    do Y .......
else
    do Z
**Example**

Test cases for:

branch coverage:

\[ a = [1, -45, 3] \]

decision coverage:

```c
1. pos_sum(a, num_of_entries, sum)
2. sum = 0
3. int i = 1
4. while (i <= num_of_entries)
5. if a[i] > 0
6. sum = sum + a[i]
7. endif
8. i = i + 1
end while
9. end pos_sum
```

if(age < 65 and married == true)
    do X
do Y
else
    do Z
Example

Test cases for:

branch coverage:
   a=[1,-45,3]

decision coverage:
   age=30, married = true

```plaintext
/* pos_sum finds the sum of all positive numbers (greater than zero) stored in an integer array a. Input parameters are num_of_entries, an integer, and a, an array of integers with num_of_entries elements. The output parameter is the integer sum. */

1. pos_sum(a, num_of_entries, sum)
2. sum = 0
3. int i = 1
4. while (i <= num_of_entries)
5.    if a[i] > 0
6.       sum = sum + a[i]
7.    endif
8.    i = i + 1
9. end while
10. end pos_sum

if(age < 65 and married == true)
    do X
    do Y
else
    do Z
```
Test cases for:

**branch coverage:**

   \[a = [1, -45, 3]\]

**decision coverage:**

   - age = 30, married = true
   - age = 70, married = true

```c
1. pos_sum(a, num_of_entries, sum)
2. sum = 0
3. int i = 1
4. while (i <= num_of_entries)
5. if a[i] > 0
6. sum = sum + a[i]
   endif
7. i = i + 1
   end while
8. end pos_sum
```

```c
if(age < 65 and married == true)
   do X
   do Y ........
else
   do Z
```
Test cases for:

**branch coverage:**
\[ a = [1, -45, 3] \]

**decision coverage:**
- age = 30, married = true
- age = 70, married = true

**condition coverage:**
- if(age < 65 and married == true) do X
doo Y .......
- else do Z
Example

Test cases for:

**branch coverage:**

\[ a = [1, -45, 3] \]

**decision coverage:**

- age = 30, married = true
- age = 70, married = true

**condition coverage:**

- age = 30, married = false

```plaintext
1. pos_sum(a, num_of_entries, sum)
2. sum = 0
3. inti = 1
4. while (i <= num_of_entries)
5.   if a[i] > 0
6.     sum = sum + a[i]
7.   endif
8.   i = i + 1
9. end while
10. end pos_sum
```

```plaintext
if(age < 65 and married = = true)
do X
   do Y .......
else
   do Z
```
Example

Test cases for:

**branch coverage:**
a=[1,-45,3]

**decision coverage:**
age=30, married = true
age=70, married = true

**condition coverage:**
age=30, married = false
age=70, married = true

```
1. pos_sum(a, num_of_entries, sum)
2. sum = 0
3. int i = 1
4. while (i <= num_of_entries)
5. if a[i] > 0
6. sum = sum + a[i]
    endif
7. i = i + 1
    end while
8. end pos_sum
```

```
if(age < 65 and married == true)
    do X
    do Y .......
else
    do Z
```
How does one design test sets with concrete coverage goals?
Automated Test Generation

We want test generation techniques that are:
Automated Test Generation

We want test generation techniques that are:

- Automatic,
We want test generation techniques that are:

- Automatic,
- Scalable,
Automated Test Generation

We want test generation techniques that are:

- Automatic,
- Scalable,
- capable of testing many properties.
Automated Test Generation

We want test generation techniques that are:

- Automatic,
- Scalable,
- capable of testing many properties.

The main challenge: given a program, automatically generate a set of inputs that upon execution will achieve a certain coverage criterion.
Supporting Technology
Supporting Technology

- Static Analysis (symbolic execution)
Supporting Technology

- Static Analysis (symbolic execution)
- Dynamic Analysis (instrumentation and runtime verification)
Supporting Technology

- **Static Analysis** (symbolic execution)
- **Dynamic Analysis** (instrumentation and runtime verification)
- **Model Checking** (systematic state-space exploration)
Supporting Technology

- Static Analysis (symbolic execution)
- Dynamic Analysis (instrumentation and runtime verification)
- Model Checking (systematic state-space exploration)
- Constraint-Solvers (SMT!)
Symbolic Execution
Example
Example

```python
if (x > y):
    x=x+y
    y=x-y
    x=x-y
    if (x - y > 0)
        assert false
return (x, y)
```
Example

if (x > y):
    x = x + y
    y = x - y
    x = x - y
    if (x - y > 0)
        assert false
    return (x, y)
Example

```python
def f(x, y):
    if (x > y):
        x = x + y
        y = x - y
        x = x - y
        if (x - y > 0):
            assert false
    return (x, y)
```

**Symbolic state** maps variables to symbolic values.

**Path condition** is a quantifier-free formula over the symbolic inputs that encodes all branch decisions taken so far.
Example

```python
def f(x, y):
    if (x > y):
        x = x + y
        y = x - y
        x = x - y
        if (x - y > 0):
            assert false
    return (x, y)
```

Symbolic state maps variables to symbolic values. Path condition is a quantifier-free formula over the symbolic inputs that encodes all branch decisions taken so far. All paths in the program form its execution tree, in which some paths are feasible and some are infeasible.
Example

```python
def f(x, y):
    if (x > y):
        x = x + y
        y = x - y
        x = x - y
    if (x - y > 0):
        assert false
    return (x, y)
```
Example

if (x > y):
    x=x+y
    y=x-y
    x=x-y
    if (x - y > 0)
        assert false
    return (x, y)
Example

if (x > y):
    x = x + y
    y = x - y
    x = x - y
    if (x - y > 0):
        assert false
    return (x, y)
Example

```python
if (x > y):
    x = x + y
    y = x - y
    x = x - y
    if (x - y > 0)
        assert false
return (x, y)
```
Example

```python
if (x > y):
    x=x+y
    y=x-y
    x=x-y
    if (x - y > 0):
        assert false
    return (x, y)
```

![Diagram of example execution](diagram.png)
Example

if (x > y):
    x=x+y
    y=x-y
    x=x-y
    if (x - y > 0)
        assert false
    return (x, y)
Example

if (x > y):
    x=x+y
    y=x-y
    x=x-y
    if (x - y > 0)
        assert false
return (x, y)

Classic symbolic execution

```python
def f(x, y):
    if (x > y):
        x = x + y
        y = x - y
        if (x - y > 0):
            assert false
        return (x, y)
    x = x - y
    y = x - y
    if (x - y > 0):
        assert false
    return (x, y)
```

Symbolic state

execute the program on infeasible symbolic inputs that encodes all branch decisions taken so far.

Path condition

is a quantifier-free formula over maps variables to symbolic values.

Y - X > 0
Y - X ≤ 0

infeasible feasible
Symbolic Execution Summary
Symbolic Execution Summary

- A static analysis technique
Symbolic Execution Summary

- A static analysis technique
- Symbolic values instead of concrete inputs.
Symbolic Execution Summary

- A static analysis technique
- Symbolic values instead of concrete inputs.
- At each program location, the state is defined by:
Symbolic Execution Summary

- A static analysis technique
- Symbolic values instead of concrete inputs.
- At each program location, the state is defined by:
  - current assignments to symbolic values and local variables.
Symbolic Execution Summary

- A static analysis technique
- Symbolic values instead of concrete inputs.
- At each program location, the state is defined by:
  - current assignments to symbolic values and local variables.
  - a path condition that must hold for the execution to reach that location (conditions on the inputs to reach the location).
Symbolic Execution Summary

- A static analysis technique
- Symbolic values instead of concrete inputs.

At each program location, the state is defined by:

- current assignments to symbolic values and local variables.
- a path condition that must hold for the execution to reach that location (conditions on the inputs to reach the location).

At each branch, both paths are followed.
Symbolic Execution Summary

- A **static analysis** technique

- **Symbolic values** instead of concrete inputs.

- At each program location, the **state** is defined by:
  - current **assignments** to symbolic values and local variables.
  - a **path condition** that must hold for the execution to reach that location (conditions on the inputs to reach the location).

- At each branch, both paths are followed.

  - On the true branch: the condition is added to the path constraints.
Symbolic Execution Summary

- A static analysis technique

- Symbolic values instead of concrete inputs.

- At each program location, the state is defined by:
  - current assignments to symbolic values and local variables.
  - a path condition that must hold for the execution to reach that location (conditions on the inputs to reach the location).

- At each branch, both paths are followed.
  - On the true branch: the condition is added to the path constraints.
  - On the false branch: the negation of the condition is added.
Symbolic Execution Summary

- A **static analysis** technique
- **Symbolic values** instead of concrete inputs.
- At each program location, **the state** is defined by:
  - current **assignments** to symbolic values and local variables.
  - a **path condition** that must hold for the execution to reach that location (conditions on the inputs to reach the location).
- At each branch, **both paths are followed**.
  - On the true branch: the condition is added to the path constraints.
  - On the false branch: the negation of the condition is added.
- If the branch is infeasible, execution stops.
Symbolic Execution Summary

- A **static analysis** technique
- **Symbolic values** instead of concrete inputs.
- At each program location, **the state** is defined by:
  - current **assignments** to symbolic values and local variables.
  - a **path condition** that must hold for the execution to reach that location (conditions on the inputs to reach the location).
- At each branch, **both paths are followed**.
  - On the true branch: the condition is added to the path constraints.
  - On the false branch: the negation of the condition is added.
- If the branch is infeasible, execution stops.

Note: here, the goal is path coverage.
Automation of Symbolic Execution
Automation of Symbolic Execution

- Static Analysis is used to collect path conditions.
Automation of Symbolic Execution

- **Static Analysis** is used to collect path conditions.

- **Decision procedures** are used to check the feasibility of path constraints.
Automation of Symbolic Execution

- Static Analysis is used to collect path conditions.
- Decision procedures are used to check the feasibility of path constraints.
- Constraint solving is used to get concrete inputs for testing.
Loops and Recursion

Loops and Recursion: infinite execution tree
Loops and Recursion: infinite execution tree

```plaintext
init;
while (C) {
    B;
}
assert P;
```
Loops and Recursion

Loops and Recursion: infinite execution tree

```
init;
while (C) {
    B;
}
assert P;
```

Infinite Execution Tree
Dealing with infinite execution trees:

- Finitize paths by limiting the size of PCs (bounded verification)
- Use loop invariants (verification)

```
init;
while (C) {
    B;
}
assert P;
```

Loop Invariant: $I$

```
init;
assert I;
makeSymbolic(targets(B));
assume I;
if (C) {
    B;
    assert I;
} else
    assert P;
```
What is difficult about Symbolic Execution?

Loops and Recursion: infinite execution tree

Path Explosion: exponentially many paths
What is difficult about Symbolic Execution?

Loops and Recursion: infinite execution tree

Path Explosion: exponentially many paths

If the space is too large to fully cover using symbolic execution, a mix of random testing and symbolic execution can help achieve some coverage.
What is difficult about Symbolic Execution?

Loops and Recursion: infinite execution tree

Path Explosion: exponentially many paths

Solver Limitations: undecidable theories
What is difficult about Symbolic Execution?

Loops and Recursion: infinite execution tree

Path Explosion: exponentially many paths

Solver Limitations: undecidable theories

Environment Modelling: native/system/library calls
What is difficult about Symbolic Execution?

Loops and Recursion: infinite execution tree

Path Explosion: exponentially many paths

Solver Limitations: undecidable theories

Environment Modelling: native/system/library calls

```c
int obscure(int x, int y) {
    if (x == hash(y)) return -1; // error
    return 0; // ok
}
```
What is difficult about Symbolic Execution?

Loops and Recursion: infinite execution tree

Path Explosion: exponentially many paths

Solver Limitations: undecidable theories

Environment Modelling: native/system/library calls

```c
int obscure(int x, int y) {
    if (x == hash(y)) return -1; // error
    return 0; // ok
}
```

Code not available, so impossible to do SE!
What is difficult about Symbolic Execution?

**Loops and Recursion:** infinite execution tree

**Path Explosion:** exponentially many paths

**Solver Limitations:** undecidable theories

**Environment Modelling:** native/system/library calls

**Heap Modeling:** pointers
What is difficult about Symbolic Execution?

Loops and Recursion: infinite execution tree

Path Explosion: exponentially many paths

Solver Limitations: undecidable theories

Environment Modelling: native/systemlibrary calls

Heap Modeling: pointers

```java
public void foo(Node n1, Node n2) {
    if (n1 != null && n2 != null) {
        n1.x = 5;
        n2.x = 6;
        assert n1.x == 5 && n2.x == 6;
    }
}
```
What is difficult about Symbolic Execution?

Loops and Recursion: infinite execution tree

Path Explosion: exponentially many paths

Solver Limitations: undecidable theories

Environment Modelling: native/system/library calls

Heap Modeling: pointers

The famous **aliasing** problem

```java
public void foo(Node n1, Node n2) {
    if (n1 != null && n2 != null) {
        n1.x = 5;
        n2.x = 6;
        assert n1.x == 5 && n2.x == 6;
    }
}
```
Lazy Concretization
Example

class Node {
    int elem;
    Node next;
}

n = symbolic(Node);
x = n.next;
Example

class Node {
    int elem;
    Node next;
}

n = symbolic(Node);
x = n.next;
```java
class Node {
    int elem;
    Node next;
}

n = symbolic(Node);
x = n.next;
```
```java
class Node {
    int elem;
    Node next;
}

n = symbolic(Node);
x = n.next;
```

---

**Example**

![Diagram](image.png)

```
class Node {
    int elem;
    Node next;
}

n = symbolic(Node);
x = n.next;
```
```java
class Node {
    int elem;
    Node next;
}

n = symbolic(Node);
x = n.next;
```
Concolic Execution
Example

typedef struct cell {
  int v;
  struct cell *next;
} cell;

int f(int v) {
  return 2*v + 1;
}

int testme(cell *p, int x) {
  if (x > 0)
    if (p != NULL)
      if (f(x) == p->v)
        if (p->next == p)
          abort();
  return 0;
}
typedef struct cell {
    int v;
    struct cell *next;
} cell;

int f(int v) {
    return 2*v + 1;
}

int testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (f(x) == p->v)
                if (p->next == p)
                    abort();
    return 0;
}
Example

typedef struct cell {
    int v;
    struct cell *next;
} cell;

int f(int v) {
    return 2*v + 1;
}

int testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (f(x) == p->v)
                if (p->next == p)
                    abort();
    return 0;
}

Concrete | PC
---------|----
p ⇔ null  | x > 0 ∧ p=null
x ⇔ 236   |

A0
next: null
v: 634

A0
p ⇔ A0  | x > 0 ∧ p≠null ∧ p.v ≠ 2x + 1
x ⇔ 236   |

Execute concretely and symbolically.

Negate last decision and solve for new inputs.

next: null
p: A0
v: 634
Example

typedef struct cell {
    int v;
    struct cell *next;
} cell;

int f(int v) {
    return 2*v + 1;
}

int testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (f(x) == p->v)
                if (p->next == p)
                    abort();
    return 0;
}

Concrete | PC
---|---
p \mapsto \text{null} | x > 0 \land p=\text{null}
x \mapsto 236

A0

next: \text{null}
p \mapsto A0
v: 634
x \mapsto 236

p \mapsto A0
x > 0 \land p\neq\text{null} \land
p.v \neq 2x + 1

A0

next: \text{null}
p \mapsto A0
v: 3
x \mapsto 1

x > 0 \land p\neq\text{null} \land
p.v = 2x + 1 \land
p.next \neq p
**Example**

```c
typedef struct cell {
    int v;
    struct cell *next;
} cell;

int f(int v) {
    return 2*v + 1;
}

int testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (f(x) == p->v)
                if (p->next == p)
                    abort();
    return 0;
}
```

**Concrete:**

- `p` is null
- `x` is 236

**PC:**

- `x > 0 ∧ p=null`

**A0**

- `next: null`
- `p` is A0
- `v: 634`
- `x` is 236

**A0**

- `next: null`
- `p` is A0
- `v: 3`
- `x` is 1

**A0**

- `next: A0`
- `p` is A0
- `v: 3`
- `x` is 1

**Negate last decision and solve for new inputs:**

- `x > 0 ∧ p≠null ∧ p.v = 2x + 1`
- `p.next ≠ p`

**Concrete:**

- `p` is A0
- `x` is 1

**PC:**

- `x > 0 ∧ p≠null ∧ p.v = 2x + 1 ∧ p.next = p`
Concolic Testing Overview
Let's not build the symbolic execution tree
Concolic Testing Overview

- Let’s not build the symbolic execution tree
- Instead, we execute the program on (possibly random) concrete input values, and observe what happens.
Concolic Testing Overview

- Let’s not build the *symbolic execution tree*
- Instead, we *execute* the program on (possibly random) *concrete input values*, and observe what happens.
  - collect *symbolic constraints* at branches whenever possible
Concolic Testing Overview

- Let's not build the **symbolic execution tree**

- Instead, we **execute** the program on (possibly random) **concrete input values**, and observe what happens.
  - collect **symbolic constraints** at branches whenever possible
  - negate the last constraint to take another side of a branch
Concolic Testing Overview

- Let’s not build the **symbolic execution tree**
- Instead, we **execute** the program on (possibly random) **concrete input values**, and observe what happens.
  - collect **symbolic constraints** at branches whenever possible
  - negate the last constraint to take another side of a branch
  - call a constraint solver to generate **new test inputs**
Concolic Testing Overview

- Let’s not build the **symbolic execution tree**
- Instead, we **execute** the program on (possibly random) **concrete input values**, and observe what happens.
  - collect **symbolic constraints** at branches whenever possible
  - negate the last **constraint** to take another side of a branch
  - call a constraint solver to generate **new test inputs**
  - next **execution** will be driven by new inputs
Let’s not build the *symbolic execution tree*

Instead, we *execute* the program on (possibly random) *concrete input values*, and observe what happens.

- Collect *symbolic constraints* at branches whenever possible
- *Negate the last constraint* to take another side of a branch
- Call a constraint solver to generate *new test inputs*
- *Next execution* will be driven by new inputs
- Check that the new branch is taken
Concolic Testing Overview

- Let’s not build the **symbolic execution tree**
- Instead, we **execute** the program on (possibly random) **concrete input values**, and observe what happens.
  - collect **symbolic constraints** at branches whenever possible
  - negate the last constraint to take another side of a branch
  - call a constraint solver to generate **new test inputs**
  - next execution will be driven by new inputs
  - check that the new branch is taken
- Repeat until **all paths are covered** (or some other coverage goal is achieved): may not terminate!
Example

```c
void test(int x, int y) {
    if (x > 0) {
        if (y == hash(x))
            S0;
        else
            S1;
        if (x > 3 && y > 10)
            S3;
        else
            S4;
    }
}
	native int hash(x) {
        if (0<=x<=10) return x*10;
        else return 0;
    }
```

Concolic' Execution
Example

```c
void test(int x, int y) {
    if (x > 0) {
        if (y == hash(x))
            S0;
        else
            S1;
        if (x > 3 && y > 10)
            S3;
        else
            S4;
    }
}

native int hash(x) {
    if (0<=x<=10) return x*10;
    else return 0;
}
```
Example

```c
void test(int x, int y) {
    if (x > 0) {
        if (y == hash(x))
            S0;
        else
            S1;
        if (x > 3 && y > 10)
            S3;
        else
            S4;
    }
}

native int hash(x) {
    if (0<=x<=10) return x*10;
    else return 0;
}
```

Example

```c
void test(int x, int y) {
    if (x > 0) {
        if (y == hash(x))
            S0;
        else
            S1;
        if (x > 3 && y > 10)
            S3;
        else
            S4;
    }
}

native int hash(x) {
    if (0<=x<=10) return x*10;
    else return 0;
}
```

Concolic Execution

test(4,11) [X>0] [X>0 & Y!=40 & X>3 & Y>10] [X>0 & Y!=40 & X>3 & Y>10]

test(4,40) [X>0] [X>0 & Y=40] [X>0 & Y=40] [X>0 & Y=40 & X>3 & Y>10] [X>0 & Y=40 & X>3 & Y>10] [X>0 & Y=40 & X>3 & Y>10]
Solver-related Example

```c
void foo(int x, int y) {
    int z = x * x * x; /* could be z = h(x) */
    if (z == y) {
        abort(); /* error */
    }
}
```

- Assume we can reason about linear constraints only
- Initially \( x = 3 \) and \( y = 7 \) (randomly generated)
- Concrete \( z = 27 \), but symbolic \( z = x * x * x \)
  - Cannot handle symbolic value of \( z \! \)
  - Stuck?
void foo(int x, int y) {
    int z = x*x*x; /* could be z = h(x) */
    if (z == y) {
        abort(); /* error */
    }
}

- Assume we can reason about linear constraints only
- Initially x = 3 and y = 7 (randomly generated)
- Concrete z = 27, but symbolic z = x*x*x
  - Cannot handle symbolic value of z!
  - Stuck?
  - **NO!** Use concrete value z = 27 and proceed...
void foo(int x,int y){
    int z = x*x*x; /* could be z = h(x) */
    if (z == y) {
        abort(); /* error */
    }
}

- Assume we can reason about linear constraints only
- Initially x = 3 and y = 7 (randomly generated)
- Concrete z = 27, but symbolic z = x*x*x
  - Cannot handle symbolic value of z!
  - Stuck?
  - NO! Use concrete value z = 27 and proceed…

Replace symbolic expression by concrete value when symbolic expression becomes unmanageable (e.g. non-linear)
void foo(int x, int y) {
    int z = x * x * x; /* could be z = h(x) */
    if (z == y) {
        abort(); /* error */
    }
}

- Assume we can reason about linear constraints only
- Initially x = 3 and y = 7 (randomly generated)
  - Concrete z = 27, but symbolic z = x * x * x  
    - Cannot handle symbolic value of z!
    - Stuck?
    - **NO!** Use concrete value z = 27 and proceed...
- Take else branch with constraint 27 != y
- Solve 27 = y to take then branch
- Execute next run with x = 3 and y = 27

**Replace** symbolic expression by **concrete value** when symbolic expression becomes unmanageable (e.g. non-linear)
Examples of Sophisticated Testing Tools
Examples of Sophisticated Testing Tools

- JPF: emerged from being an explicit-state model checker
Examples of Sophisticated Testing Tools

- **JPF**: emerged from being an explicit-state model checker
- **Sage**: Microsoft’s test generation tool
Examples of Sophisticated Testing Tools

- **JPF**: emerged from being an explicit-state model checker
- **Sage**: Microsoft’s test generation tool
- **PEX**: unit tests
Examples of Sophisticated Testing Tools

- **JPF**: emerged from being an explicit-state model checker
- **Sage**: Microsoft’s test generation tool
- **PEX**: unit tests
- **Yogi**: attempts to simultaneously prove the program correct or find a counterexample (Microsoft research)
Examples of Sophisticated Testing Tools

- **JPF**: emerged from being an explicit-state model checker
- **Sage**: Microsoft’s test generation tool
- **PEX**: unit tests
- **Yogi**: attempts to simultaneously prove the program correct or find a counterexample (Microsoft Research)
- **Thread Checker**: Intel’s tool for parallel programming bugs
Examples of Sophisticated Testing Tools

- **JPF**: emerged from being an explicit-state model checker
- **Sage**: Microsoft’s test generation tool
- **PEX**: unit tests
- **Yogi**: attempts to simultaneously prove the program correct or find a counterexample (Microsoft research)
- **Thread Checker**: Intel’s tool for parallel programming bugs
- and many more …