Static Analysis

Announcements

- Project
  - part 2 feedback has been added to your google docs
  - Part 3 instructions posted
- No class Wednesday 4/30

Overview

- Evosuite Review
- Static analysis
  - Reachability analysis

Test Cluster - Initial Step

The **test cluster** is created once up-front deterministically.

Contains three sets:

1. Test Methods - all public methods on the target class and superclasses
2. Generators - all constructors and factory methods for relevant types
3. Modifiers - impure methods on the target class

Building blocks of our tests!

EvoSuite Review

Algorithm 1 Evolutionary Algorithm

```
Initialize population P with random solutions
Evaluate fitness of each individual in P
while search budget not exhausted do
    Select individuals from P based on fitness
    Apply crossover to produce new offspring
    Apply mutation to offspring
    Evaluate fitness of offspring
    Add offspring to the population P
end while
```

Creating the initial population:

1. Randomly select a constructor or method to call on the class under test
   a. Selected from the test cluster's **test methods** set

2. If the method / constructor has arguments, it needs to create those types
   a. re-use existing values in the test or create a new variable

3. We also need a target type (OOP!)

Chart 1

```
public CategoryItemLabelGenerator getItemLabelGenerator(int row,  int column, boolean selected) {
    CategoryItemLabelGenerator generator = (CategoryItemLabelGenerator)
        this.itemLabelGeneratorList.get(row);
    if (generator == null) {
        generator = this.baseItemLabelGenerator;
    }
    return generator;
}
```

How do we create primitive arguments?

```
public void randomize() {
        if (Randomness.nextDouble() >= Properties.PRIMITIVE_POOL) { //default of .5
                value = (int)(Randomness.nextGaussian() * Properties.MAX_INT) ;
        } else {
                ConstantPool constantPool = ConstantPoolManager.getInstance().getConstantPool();
                value = constantPool.getRandomInt();
        }
}
```

Constant Pool

- EvoSuite generates a “constant pool” for each primitive type

- Contains:
  - Common values (ex. Int pool has 0, 1, -1)
  - All literals in the target class

- When generating a primitive, EvoSuite either selects from this pool or generates it entirely randomly (50% default bias)

Crossover

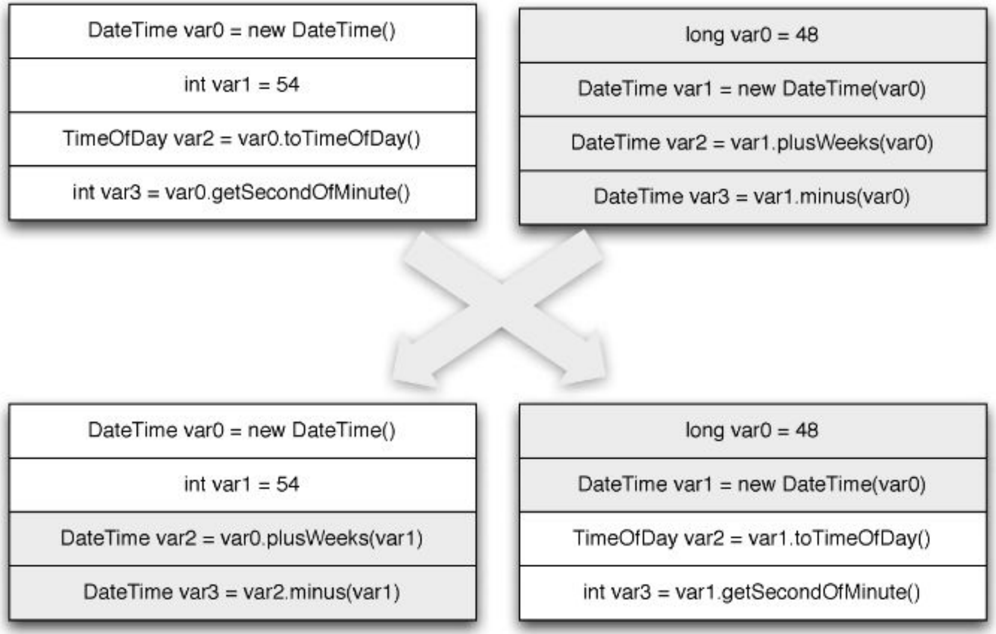


Fig. 4. Crossover between two test cases.

Mutation

1. Delete a statement
2. Insert a method call
3. Modify an existing statement
    a. Change callee (target object)
    b. Change params
    c. Change method / constructor - replace with one of the same return type
    d. Change field
    e. Change primitive

Assertions

Test suites contain a prefix and assertion

The GA we defined only generates a prefix. How does it find the assertion?

EvoSuite generates ***regression*** tests.

EvoSuite Review

Algorithm 1 Evolutionary Algorithm

```
Initialize population P with random solutions
Evaluate fitness of each individual in P
while search budget not exhausted do
    Select individuals from P based on fitness
    Apply crossover to produce new offspring
    Apply mutation to offspring
    Evaluate fitness of offspring
    Add offspring to the population P
end while
```

Seeding - Instead of generating the initial population randomly, start with a known good test

Where do these come from?

1. Usages of the MUT in the project
2. Developer written test
3. LLM generated
4. IR?
5. ...

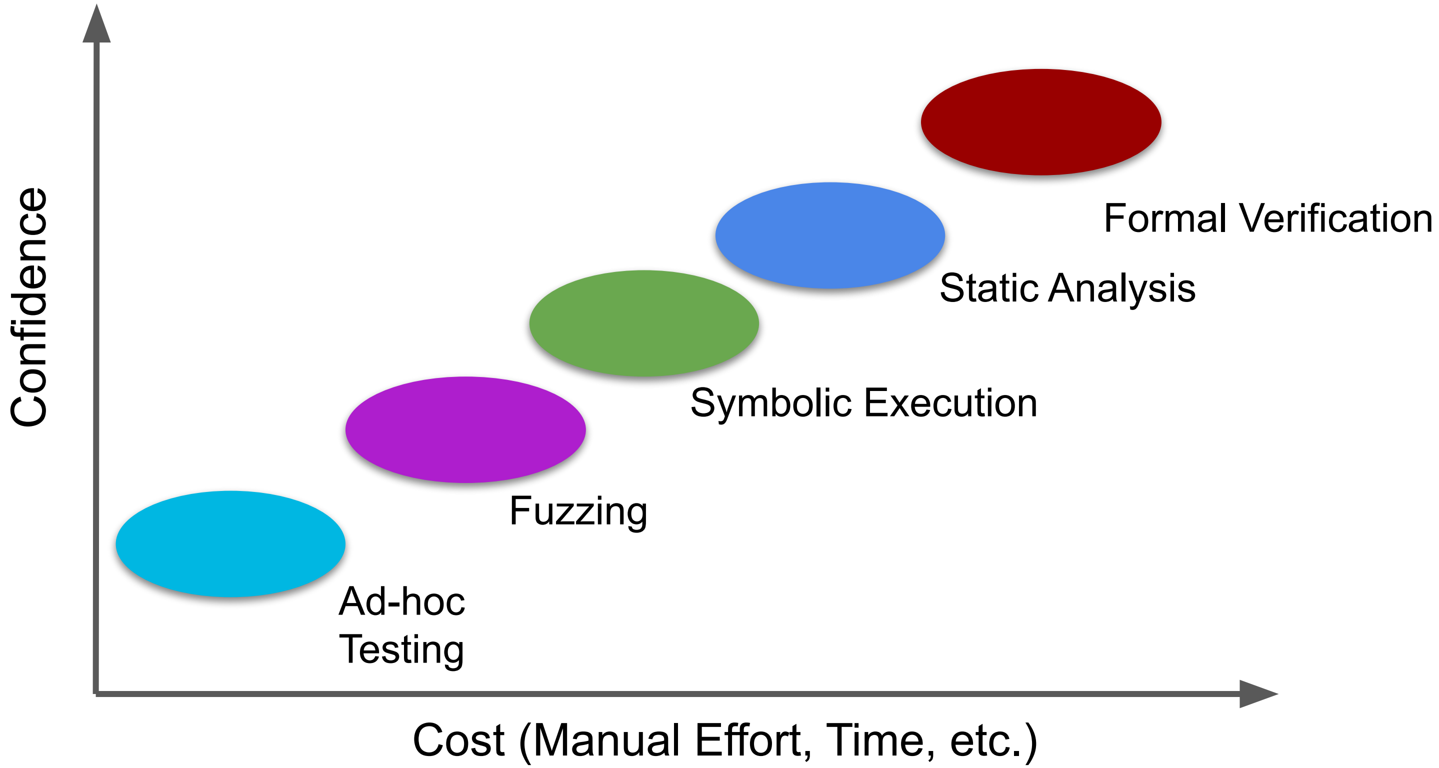
EditAs2 Review

- Step 1: Given a test prefix and the method under test, retrieve the most similar test and its assertion from a test corpus

- Step 2: Given the original (test prefix, MUT) pair and the retrieved (test prefix, MUT, assertion) tuple, perform a neural edit to the assertion

Static Analysis

Landscape of Program Analysis Techniques



What makes a good program analysis?

- Soundness:
  - If there is a bug, it will report it
  - If the tool says SAFE for some property, the program will be safe
  - (only as good as the property)
- Completeness:
  - If it reports a bug, the bug exists

	Complete	Incomplete
Sound		
Unsound		

Where do the techniques we've seen fall in this chart?

Static Analysis

- Analyzes a program without executing it

- Based on Abstract Interpretation
  - A theory of static analysis that guarantees sound outputs

- Linters are a type of static analysis

Dataflow Analysis

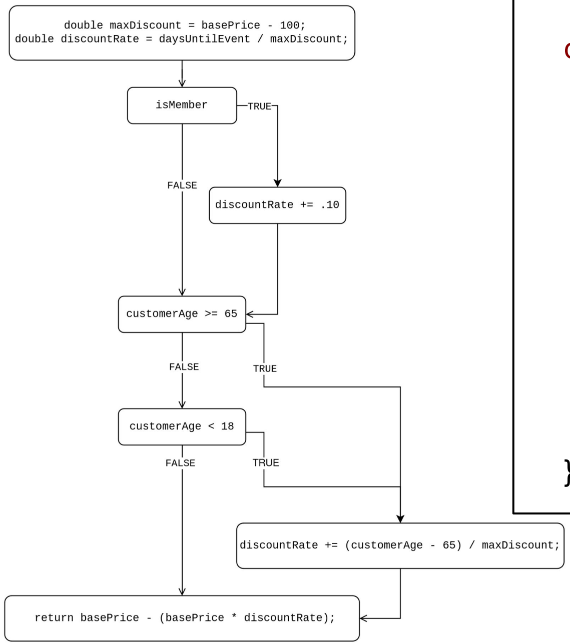
Dataflow Analysis applications

- Compilers and IDEs use data flow analysis to perform optimizations
  - Deadcode eliminations
  - Constant propagation
    - If we know x is always equal to 5 at if (x == 5) we can optimize away the check
  - Common subexpression elimination
    - Reuses previously computed statements

- Taint analysis
  - Tracks whether untrusted or sensitive input can reach sensitive operations
  - SQL injections
  - Log4J bug

Review: Control Flow Graphs (CFG)

A CFG is a directed graph with each node representing groups of one or statements and edges representing flows between them.




```
double calculatePrice(int basePrice, int daysUntilEvent, int customerAge, boolean isMember) {
    double maxDiscount = basePrice - 100;

    double discountRate = daysUntilEvent / maxDiscount;

    if (isMember) discountRate += .10;

    if (customerAge >= 65 || customerAge < 18) {
        discountRate += (customerAge - 65) / maxDiscount;
    }

    return basePrice - (basePrice * discountRate);
}
```

Reaching Definitions

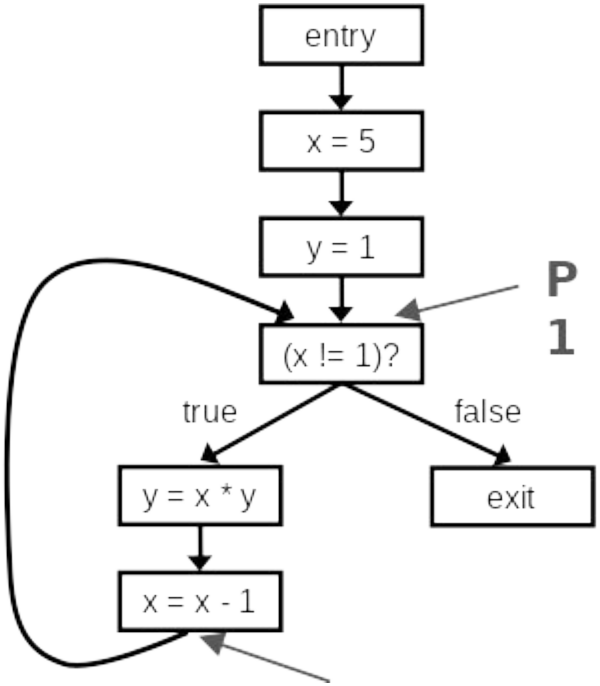
- A type of dataflow analysis
- Determine, for each program point, which assignments have been made and not overwritten, when execution reaches that point along some path
- What could this be used for?

Quiz: Reachability - which may be true?

1. The assignment y = 1 reaches P1

2. The assignment y = 1 reaches P2

3. The assignment y = x * y reaches P1



Reachability Analysis

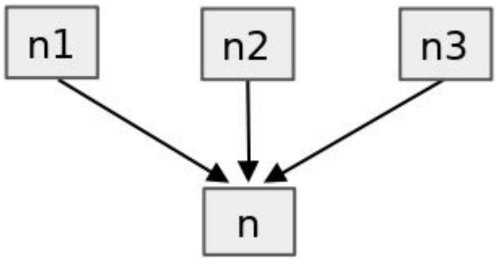
At each program point, keep a set of program facts

- Give a distinct label n to each node
- IN[n] = set of facts at entry of node n
- OUT[n] = set of facts at exit of node n
- Dataflow analysis computes IN[n] and OUT[n] for each node
- Repeat two operations until IN[n] and OUT[n] stop changing
  - Called "saturated" or "fixed point"

Reachability Analysis

- For any dataflow analysis, we must define *transfer functions* for how IN and OUT change

$$IN[n] = \bigcup_{n' \in \text{predecessors}(n)} OUT[n']$$



$$IN[n] = OUT[n1] \cup OUT[n2] \cup OUT[n3]$$

Reachability Analysis

- For any dataflow analysis, we must define *transfer functions* for how IN and OUT change

$$OUT[n] = (IN[n] - KILL[n]) \cup GEN[n]$$

n: $\boxed{\text{b ?}}$

$$GEN[n] = \varnothing$$
$$KILL[n] = \varnothing$$

n: $\boxed{x = a}$

$$GEN[n] = \{ \langle x, n \rangle \}$$
$$KILL[n] = \{ \langle x, m \rangle : m \mathrel{!=} n \}$$

Worklist Algorithm

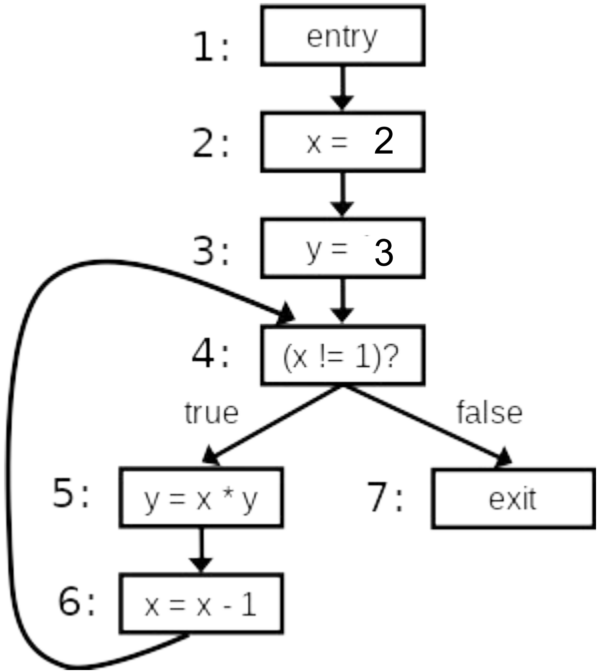
for (each node n):
    $IN[n] = OUT[n] = \varnothing$

repeat:
    for (each node n):
        $$IN[n] = \bigcup_{n' \in \text{predecessors}(n)} OUT[n']$$

        $$OUT[n] = (IN[n] - KILL[n]) \cup GEN[n]$$

until IN[n] and OUT[n] stop changing for all n

Worklist Algorithm



Iter 1:

IN[1] = ∅, OUT[1] = ∅

IN[2] = ∅, OUT[2] = {x → 2}

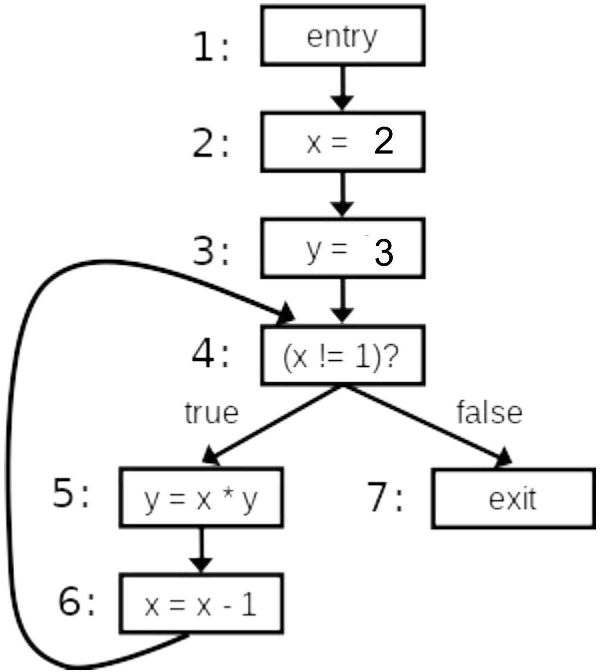
IN[3] = {x → 2}, OUT[3] = {x → 2, y → 3}

IN[4] = {x → 2, y → 3}, OUT[4] = {x → 2, y → 3}

IN[5] = {x → 2, y → 3}, OUT[5] = {x → 2, y → 6}

IN[6] = {x → 2, y → 6}, OUT[6] = {x → 1, y → 6}

Worklist Algorithm



Iter 2:

IN[1] = ∅, OUT[1] = ∅

IN[2] = ∅, OUT[2] = {x → 2}

IN[3] = {x → 2}, OUT[3] = {x → 2, y → 3}

IN[4] = [{x → 2, y → 3}, {x → 1, y → 6}]

OUT[4] = [{x → 2, y → 3}, {x → 1, y → 6}]

IN[5] = {x → 2, y → 3}, OUT[5] = {x → 2, y → 6}

IN[6] = {x → 2, y → 6}, OUT[6] = {x → 1, y → 6}

Liveness

- A variable is *live* if there is a path to a use of the variable that does not redefine the variable
- We say that a variable x is “live on exit from node j” if there is a live use of x on exit from j
- Problem statement: for each node n, compute the set of variables that are live on exit from n.

1. x=2; 2. y=4; 3. x=1; if (y>x) then 5. z=y; else 6. z=y*y; 7. x=z;

What variables are live on exit from statement 3?

Liveness Analysis

What would the gen and kill sets for liveness look like?

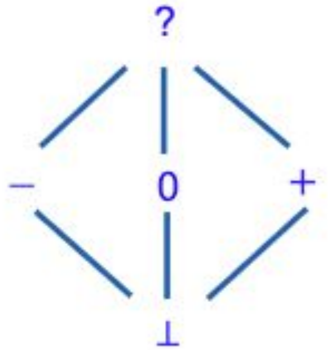
How about IN and OUT?

Static Analysis - Divide by zero bugs

Example

```
//x is a parameter

  if (x == 0) {              // p1
     x++;                     // p2

  } else if (x > 0) {        // p3

     x = x * 20;             // p4

  } else if (x < 0) {        // p5

     x = x * (-10);          // p6

  }
  assert(x > 0);
```

An **Abstract Domain** for Signs




```
//x is a parameter
  if (x == 0) {            // p1
     x++;                   // p2
  } else if (x > 0) {      // p3
     x = x * 20;           // p4
  } else if (x < 0) {      // p5
     x = x * (-10);        // p6
  }
  assert(x > 0);
```

Will this assertion always be true?

An Abstract Semantics for Signs

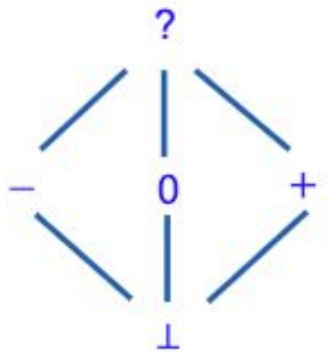
ADD	-	0	+	?
-	-	-	?	?
0	-	0	+	?
+	?	+	+	?
?	?	?	?	?

MULT	-	0	+	?
-	+	0	-	?
0	0	0	0	0
+	-	0	+	?
?	?	0	?	?

Example 2

```
if (x == 0) {       // p1
    x++;            // p2
} else if (x > 0) { // p3
    x = x * (-10);  // p4
}
assert(x != 0);
```

False Positive!



Summary

- Static analysis
  - Sound, but may have FPs
  - Ratio of acceptable FPs is 1:3

- Next class:
  - Abstract interpretation
  - Pointer analysis