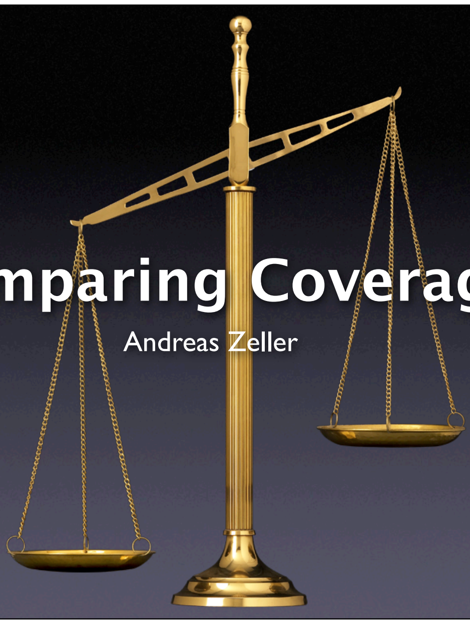
Comparing Coverage

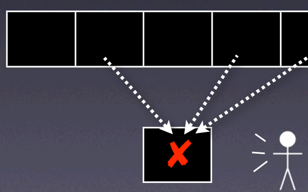


Andreas Zeller

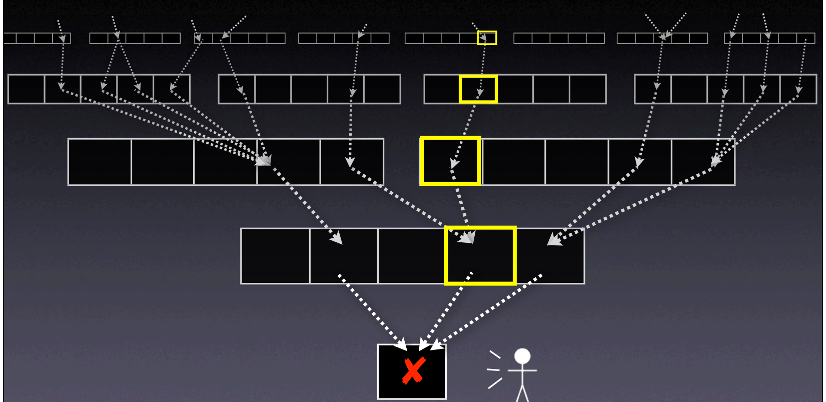
1

Tracing Infections

- For every infection, we must find the *earlier infection* that *causes* it.
- Which origin should we focus upon?

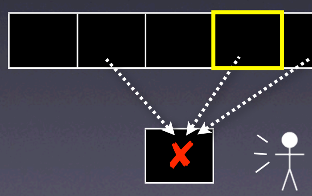


Tracing Infections



Focusing on Anomalies

- Examine origins and locations where something *abnormal* happens



4

4

What's normal?

- General idea: Use *induction* – reasoning from the particular to the general
- Start with a *multitude* of runs
- Determine *properties* that are common across all runs

5

5

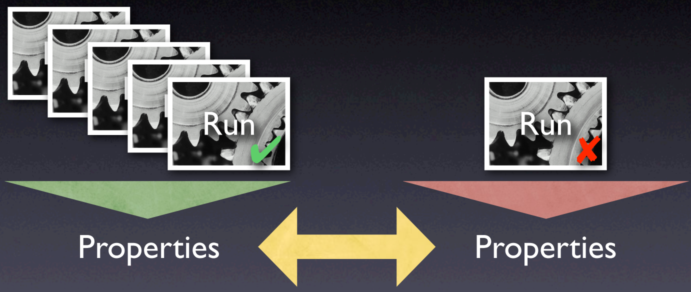
What's abnormal?

- Suppose we determine common properties of all *passing* runs.
- Now we examine a run which *fails* the test.
- Any difference in properties *correlates with failure* – and is likely to hint at failure causes

6

6

Detecting Anomalies



Differences correlate with failure

Properties

Data properties that hold in all runs:

- "At f(), x is odd"
- "$0 \leq x \leq 10$ during the run"

Code properties that hold in all runs:

- "f() is always executed"
- "After open(), we eventually have close()"

Comparing Coverage

1. Every failure is caused by an infection, which in turn is caused by a defect
2. The defect must be *executed* to start the infection
3. Code that is executed *in failing runs only* is thus likely to cause the defect

The middle program

```
$ middle 3 3 5
middle: 3

$ middle 2 1 3
middle: 1
```

10

```
int main(int arc, char *argv[])
{
    int x = atoi(argv[1]);
    int y = atoi(argv[2]);
    int z = atoi(argv[3]);
    int m = middle(x, y, z);

    printf("middle: %d\n", m);

    return 0;
}
```

11

```
int middle(int x, int y, int z) {
    int m = z;
    if (y < z) {
        if (x < y)
            m = y;
        else if (x < z)
            m = y;
    } else {
        if (x > y)
            m = y;
        else if (x > z)
            m = x;
    }
    return m;
}
```

12

Obtaining Coverage

for C programs

13

13

Obtaining Coverage

for Python programs

```
if __name__ == "__main__":
    sys.settrace(tracer)
    x = sys.argv[1]
    y = sys.argv[2]
    z = sys.argv[3]
    m = middle(x, y, z)

    print "middle:", m
```

14

14

Obtaining Coverage

for Python programs

```
def tracer(frame, event, arg):
    code = frame.f_code
    function = code.co_name
    filename = code.co_filename
    line = frame.f_lineno
    print filename + ":" + `line` + \
        ":" + function + "():", \
        event, arg
    return tracer
```

15

15

Obtaining Coverage

for Python programs

```
$ ./middle.py 3 3 5
./middle.py:13:middle(): call None
./middle.py:14:middle(): line None
./middle.py:15:middle(): line None
./middle.py:16:middle(): line None
./middle.py:18:middle(): line None
./middle.py:19:middle(): line None
./middle.py:26:middle(): line None
./middle.py:26:middle(): return 3
middle: 3
```

For remaining steps, see new project

x	3	1	3	5	5	2
y	3	2	2	5	3	1
z	5	3	1	5	4	3
`int middle(int x, int y, int z) {`	•	•	•	•	•	•
`int m = z;`	•	•	•	•	•	•
`if (y < z) {`	•	•	•	•	•	•
`if (x < y)`		•				
`m = y;`		•				
`else if (x < z)`	•				•	•
`m = y;`	•					•
`} else {`	•		•	•		
`if (x > y)`			•			
`m = y;`			•			
`else if (x > z)`						
`m = x;`						
`}`						
`return m;`	•	•	•	•	•	•
`}`	✔	✔	✔	✔	✔	✘

Discrete Coloring

- (red) executed only in failing runs
  highly suspect
- (yellow) executed in passing and failing runs
  ambiguous
- (green) executed only in passing runs
  likely correct

	x	3	1	3	5	5	2
	y	3	2	2	5	3	1
	z	5	3	1	5	4	3
`int middle(int x, int y, int z) {`		•	•	•	•	•	•
`int m = z;`		•	•	•	•	•	•
`if (y < z) {`		•	•	•	•	•	•
`if (x < y)`			•				
`m = y;`			•				
`else if (x < z)`		•				•	•
`m = y;`		•					•
`} else {`		•		•	•		
`if (x > y)`				•			
`m = y;`				•			
`else if (x > z)`							
`m = x;`							
`}`							
`return m;`		•	•	•	•	•	•
`}`		✔	✔	✔	✔	✔	✘

	x	3	1	3	5	5	2
	y	3	2	2	5	3	1
	z	5	3	1	5	4	3
`int middle(int x, int y, int z) {`		•	•	•	•	•	•
`int m = z;`		•	•	•	•	•	•
`if (y < z) {`		•	•	•	•	•	•
`if (x < y)`			•				
`m = y;`			•				
`else if (x < z)`		•				•	•
`m = y;`		•					•
`} else {`		•		•	•		
`if (x > y)`				•			
`m = y;`				•			
`else if (x > z)`							
`m = x;`							
`}`							
`return m;`		•	•	•	•	•	•
`}`		✔	✔	✔	✔	✔	✘

Continuous Coloring

- executed only in failing runs
- passing and failing runs
- executed only in passing runs

Hue

$$hue(s) = red\ hue + \frac{\%passed(s)}{\%passed(s) + \%failed(s)} \times hue\ range$$

0% passed — 100% passed

Brightness

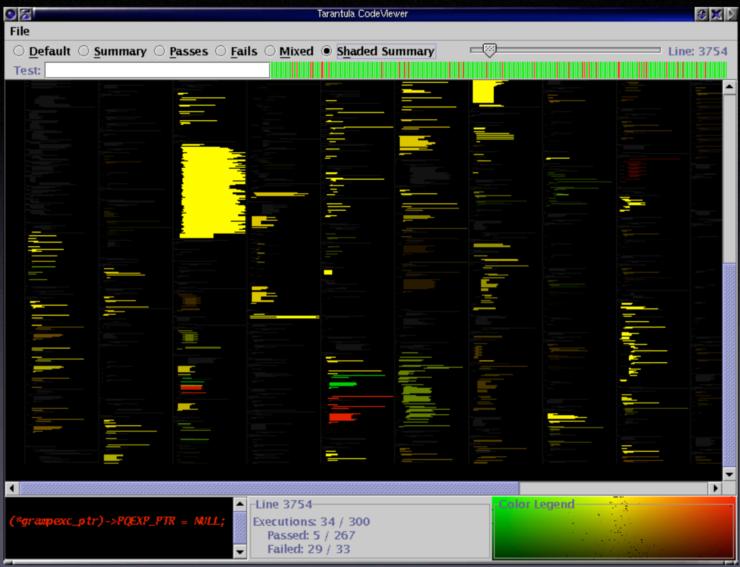
frequently executed

$$bright(s) = \max(\%passed(s), \%failed(s))$$

rarely executed

x	3	1	3	5	5	2
y	3	2	2	5	3	1
z	5	3	1	5	4	3
`int middle(int x, int y, int z) {`	•	•	•	•	•	•
`int m = z;`	•	•	•	•	•	•
`if (y < z) {`	•	•	•	•	•	•
`if (x < y)`		•				
`m = y;`		•				
`else if (x < z)`	•				•	•
`m = y;`	•					•
`} else {`	•		•	•		
`if (x > y)`			•			
`m = y;`			•			
`else if (x > z)`						
`m = x;`						
`}`						
`return m;`	•	•	•	•	•	•
`}`	✔	✔	✔	✔	✔	✘

Source: Jones et al., ICSE 2002



Source: Jones et al., ICSE 2002

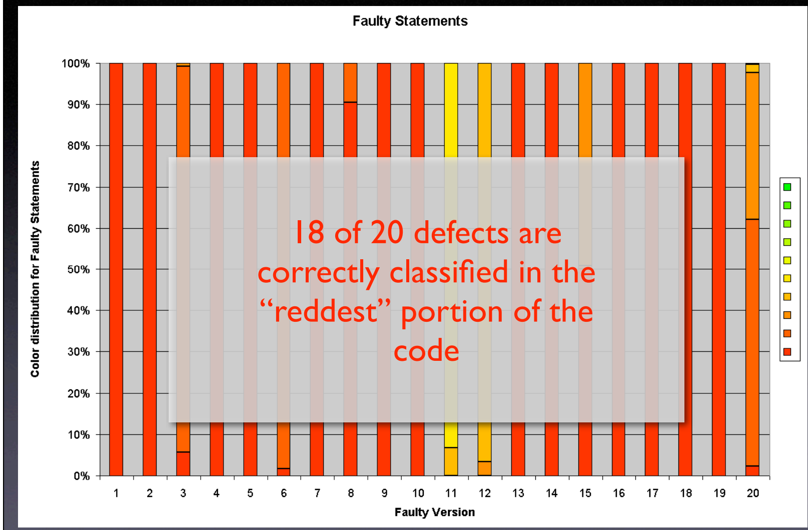
Evaluation

How well does comparing coverage detect anomalies?

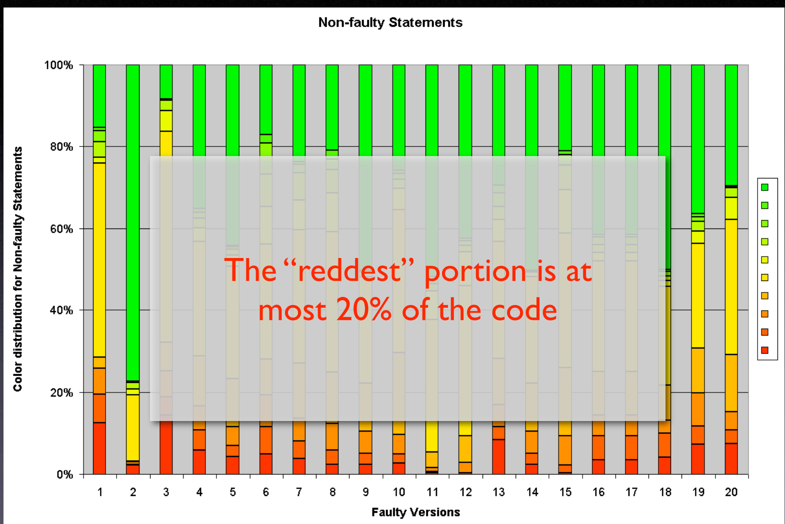
- How green are the defects? *(false negatives)*
- How red are non-defects? *(false positives)*

Space

- 8000 lines of executable code
- 1000 test suites with 156–4700 test cases
- 20 defective versions with one defect each (corrected in subsequent version)



Source: Jones et al., ICSE 2002



Source: Jones et al., ICSE 2002

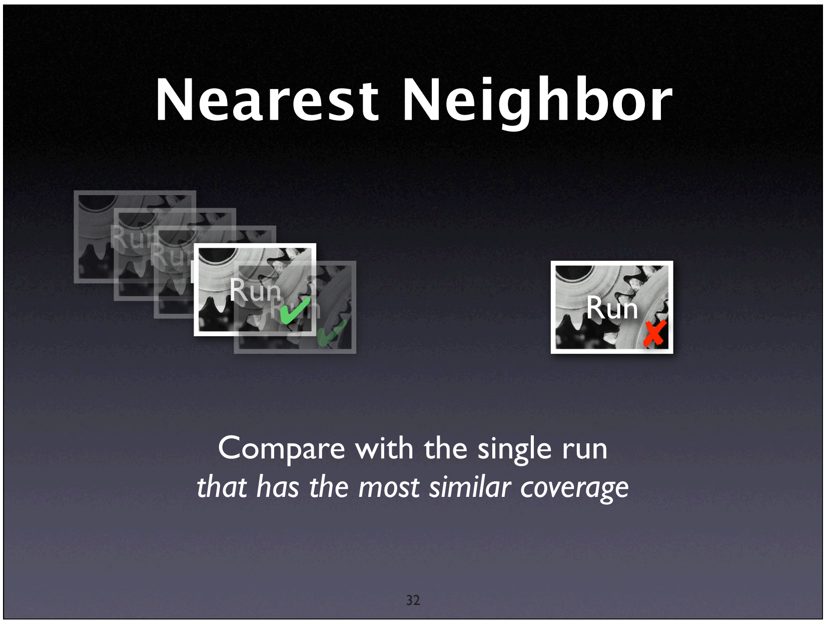
Siemens Suite

- 7 C programs, 170–560 lines
- 132 variations with one defect each
- 108 all yellow (i.e., useless)
- 1 with one red statement (at the defect)

Source: Renieris and Reiss, ASE 2003

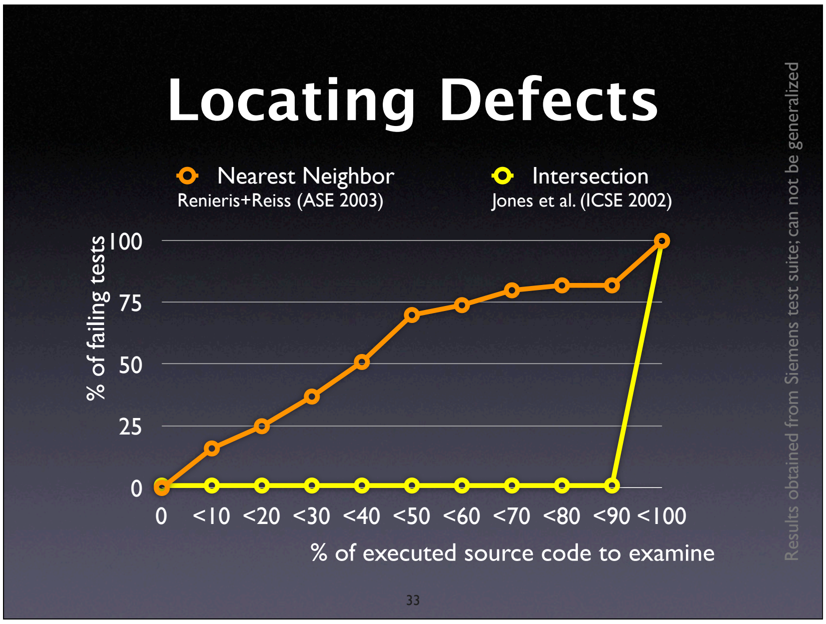
Nearest Neighbor
Run
Run
31

31



Nearest Neighbor
Run
Run
Compare with the single run
that has the most similar coverage
32

32



Locating Defects
Nearest Neighbor
Renieris+Reiss (ASE 2003)
Intersection
Jones et al. (ICSE 2002)
% of failing tests
100
75
50
25
0
0 <10 <20 <30 <40 <50 <60 <70 <80 <90 <100
% of executed source code to examine
Results obtained from Siemens test suite; can not be generalized
33

33

Concepts

- ★ Comparing coverage (or other features) shows anomalies correlated with failure
- ★ Nearest neighbor or sequences locate errors more precisely than just coverage
- ★ Models add extra program understanding
- ★ Low overhead + simple to realize