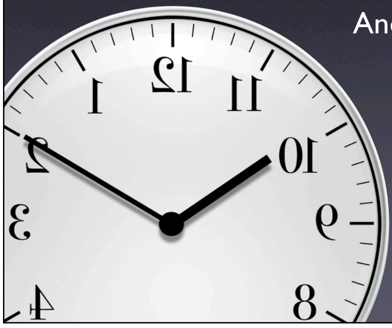
Detecting Anomalies

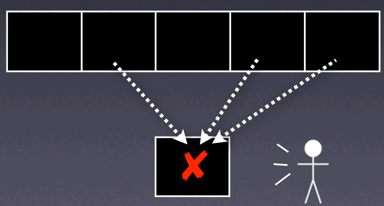
Andreas Zeller



1

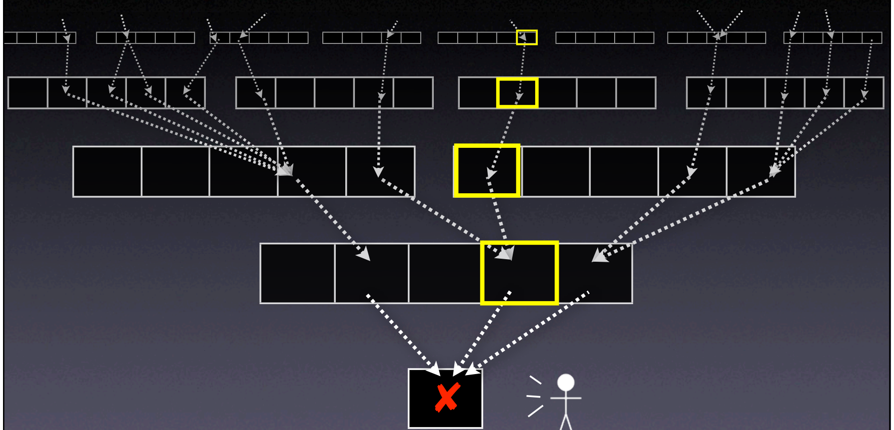
Tracing Infections

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



2

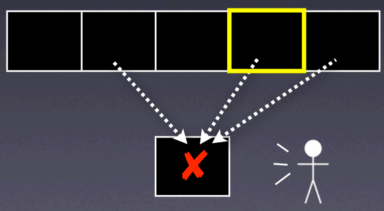
Tracing Infections



3

Focusing on Anomalies

- Examine origins and locations where something *abnormal* happens



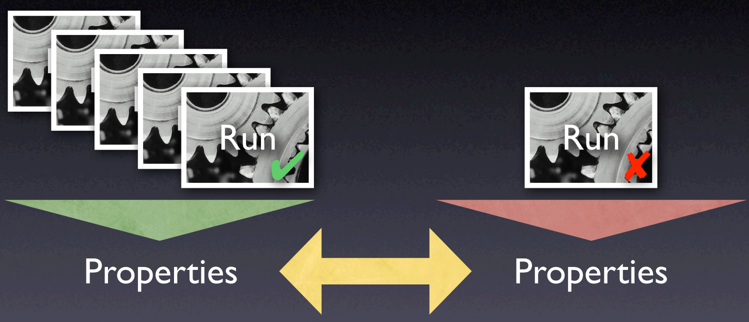
What's normal?

- 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

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

Detecting Anomalies



Differences correlate with failure

Properties

Data properties that hold in all runs:

- "At f(), x is odd"
- "$0 \le x \le 10$ 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
```



```
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;
}
```



```
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;
}
```

Obtaining Coverage

for C programs

Pippin: cgi_encode — less — 80×24

```
        4:   18:    int ok = 0;
        -:   19:
       38:   20:    while (*eptr)  /* loop to end of string ('\0' character) */
        -:   21:    {
        -:   22:        char c;
       30:   23:        c = *eptr;
       30:   24:        if (c == '+') {  /* '+' maps to blank */
        1:   25:            *dptr = ' ';
       29:   26:        } else if (c == '%') { /* '%xx' is hex for char xx */
        3:   27:            int digit_high = Hex_Values[*(++eptr)];
        3:   28:            int digit_low  = Hex_Values[*(++eptr)];
        5:   29:            if (digit_high == -1 || digit_low == -1)
        2:   30:                ok = 1; /* Bad return code */
        -:   31:            else
        1:   32:                *dptr = 16 * digit_high + digit_low;
        -:   33:        } else { /* All other characters map to themselves */
       26:   34:           *dptr = *eptr;
        -:   35:        }
       30:   36:        ++dptr; ++eptr;
        -:   37:    }
        4:   38:    *dptr = '\0';   /* Null terminator for string */
        4:   39:    return ok;
        -:   40:}
(END)
```


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

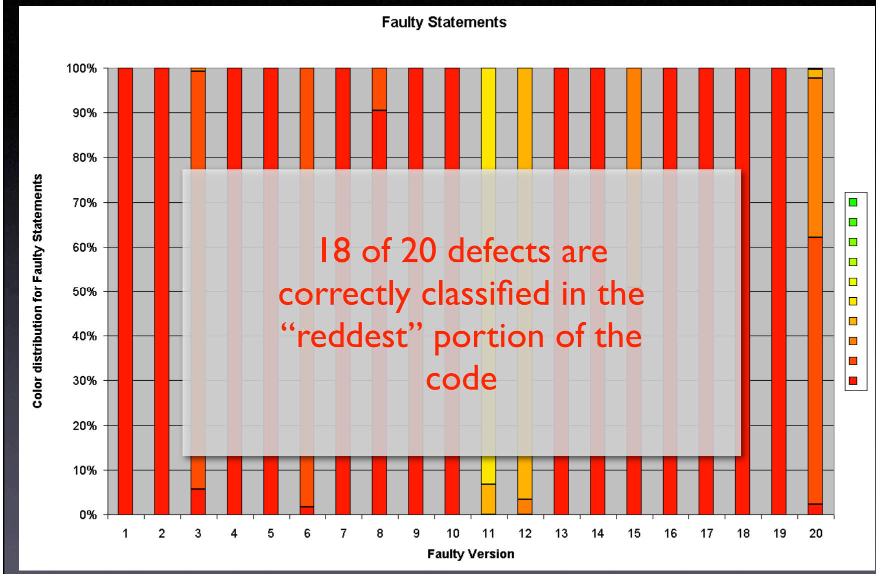
Evaluation

How well does comparing coverage detect anomalies?

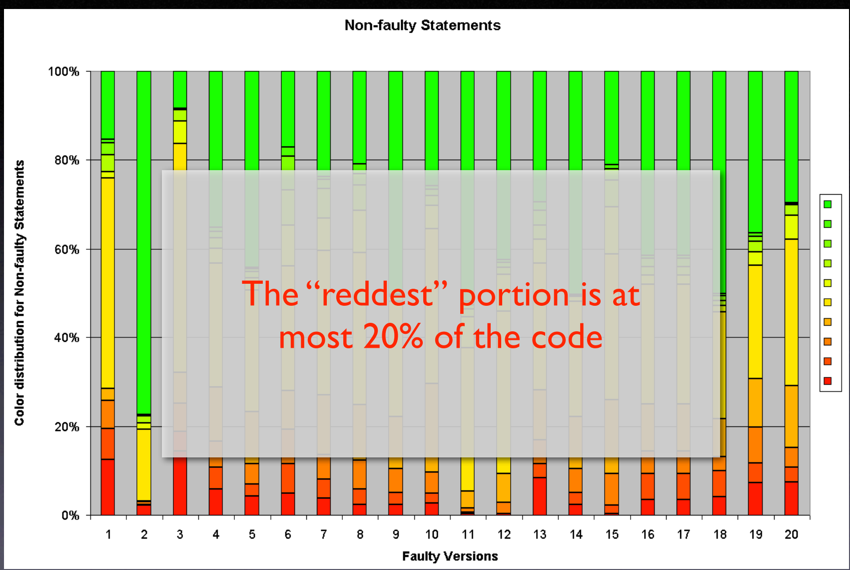
- How green are the defects? *(false negatives)*
- How red are non-defects? *(false positives)*

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

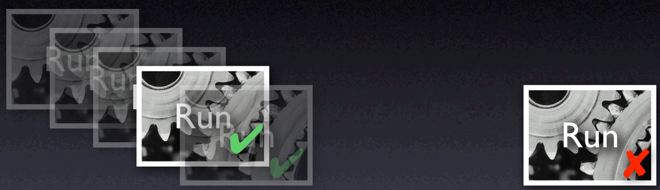
Siemens Suite

- 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

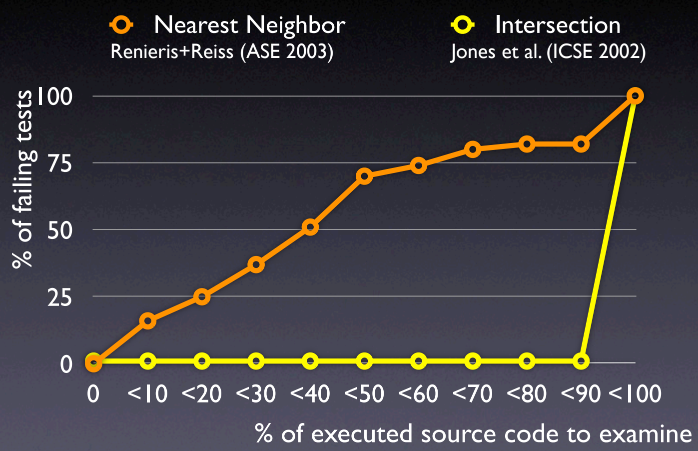
Nearest Neighbor

Nearest Neighbor



Compare with the single run
that has the most similar coverage

Locating Defects



Results obtained from Siemens test suite; can not be generalized

Sequences

Sequences of locations can correlate with failures:

open() read() close()	✔
open() close() read()	✘
close() open() read()	✘

…but all locations are executed in both runs!

The AspectJ Compiler

```
$ ajc Test3.aj
$ java test.Test3
test.Test3@b8df17.x Unexpected Signal : 11
occurred at PC=0xFA415A00
Function name=(N/A) Library=(N/A) ...
Please report this error at http://
java.sun.com/...
$
```


Coverage Differences

- Compare the failing run with passing runs
- `BcelShadow.getThisJoinPointVar()` is invoked in the failing run only
- Unfortunately, this method is correct

Sequence Differences

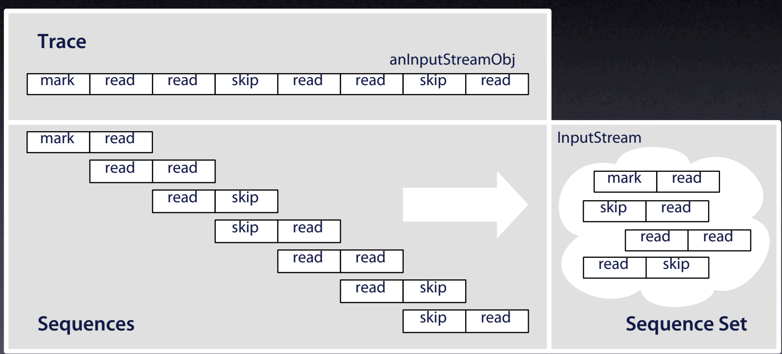
This *sequence* occurs only in the failing run:

```
⟨ ThisJoinPointVisitor.isRef(),
  ThisJoinPointVisitor.canTreatAsStatic(),
  MethodDeclaration.traverse(),
  ThisJoinPointVisitor.isRef(),
  ThisJoinPointVisitor.isRef() ⟩
```

Defect location

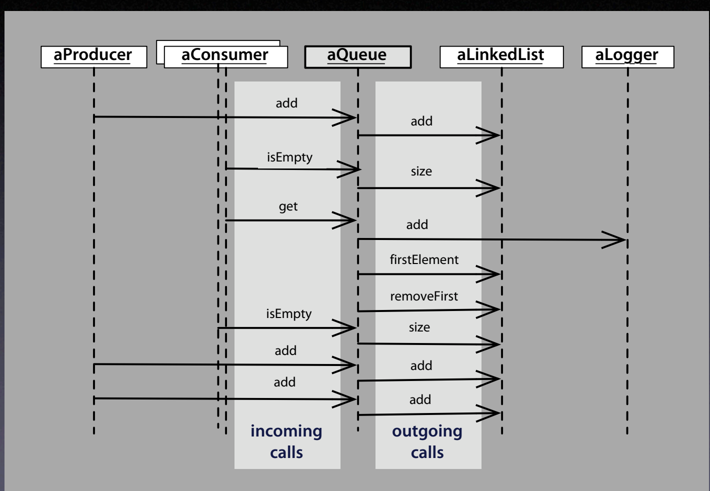
34

Collecting Sequences



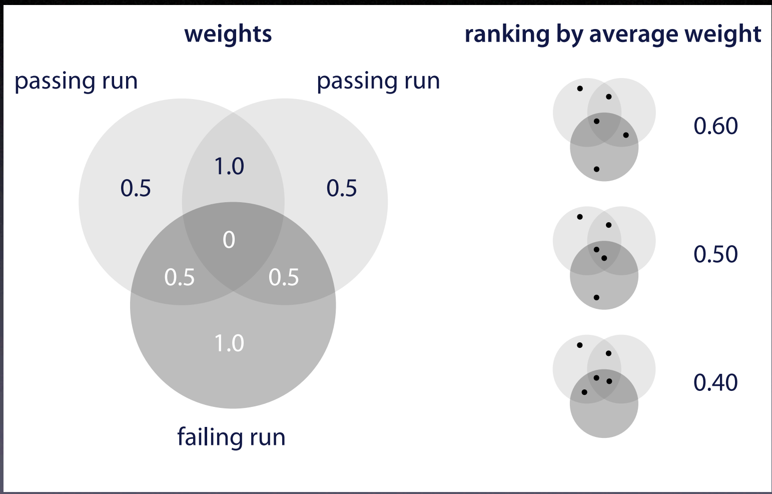
35

Ingoing vs. Outgoing



36

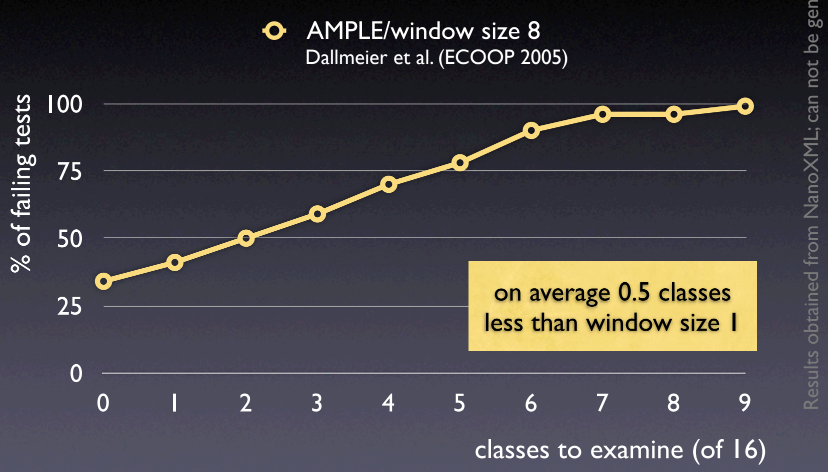
Anomalies

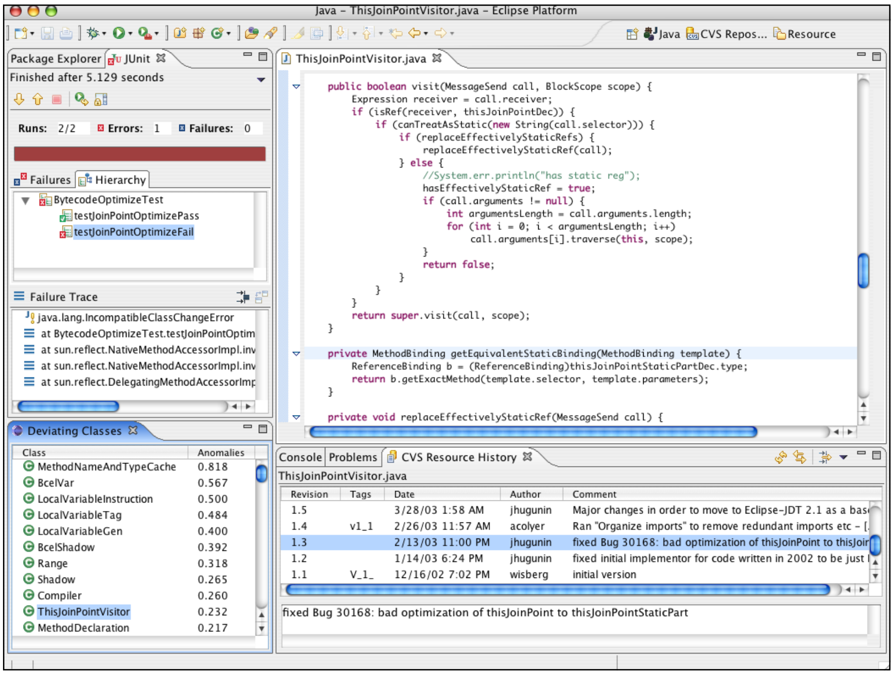


NanoXML

- Simple XML parser written in Java
- 5 revisions, each with 16–23 classes
- 33 errors discovered or seeded

Locating Defects





40

Properties

Data properties that hold in all runs:

- "At f(), x is odd"
- "0 ≤ x ≤ 10 during the run"

Code properties that hold in all runs:

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

41

Techniques

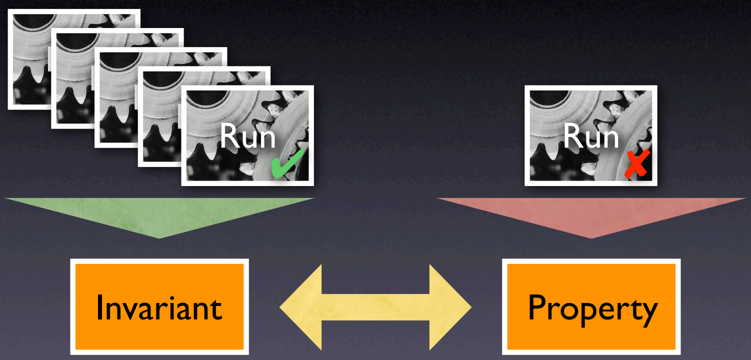
Dynamic Invariants	Value Ranges	Sampled Values

42

Techniques

Dynamic Invariants	Value Ranges	Sampled Values

Dynamic Invariants



At f(), x is odd

At f(), x = 2

Daikon

- Determines *invariants* from program runs
- Written by Michael Ernst et al. (1998–)
- C++, Java, Lisp, and other languages
- analyzed up to 13,000 lines of code

Daikon

```
public int ex1511(int[] b, int n)
{
    int s = 0;
    int i = 0;
    while (i != n) {
        s = s + b[i];
        i = i + 1;
    }
    return s;
}
```

Precondition
```
n == size(b[])
b != null
n <= 13
n >= 7
```

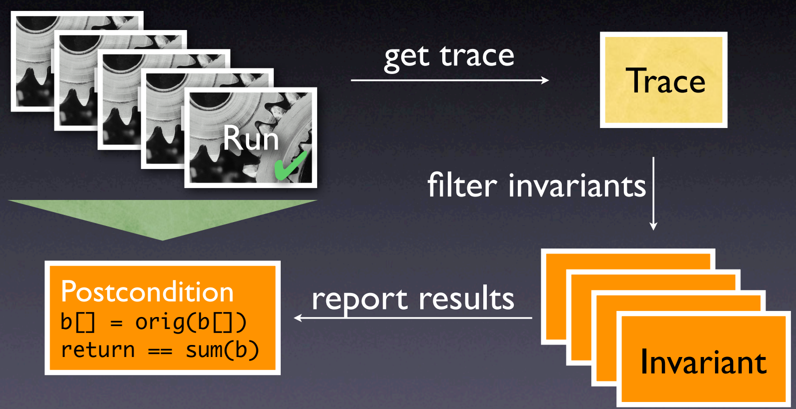
Postcondition
```
b[] = orig(b[])
return == sum(b)
```

- Run with 100 randomly generated arrays of length 7–13

46

46

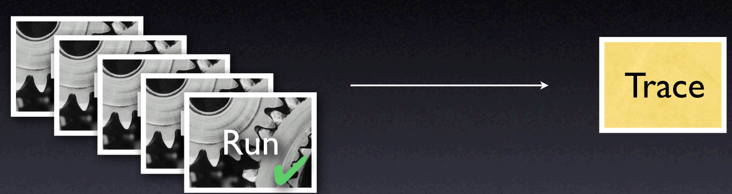
Daikon



47

47

Getting the Trace



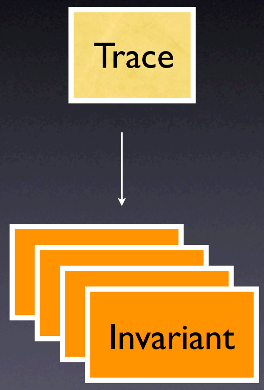
- Records all variable values at all function entries and exits
- Uses VALGRIND to create the trace

48

48

Filtering Invariants

- Daikon has a library of *invariant patterns* over variables and constants
- Only matching patterns are preserved



Method Specifications

using *primitive data*

x = 6	x ∈ {2, 5, –30}	x < y
y = 5x + 10	z = 4x +12y +3	z = fn(x, y)

using *composite data*

A subseq B	x ∈ A	sorted(A)

checked at method entry + exit

Object Invariants

string.content[string.length] = '\0'
node.left.value ≤ node.right.value
this.next.last = this

checked at entry + exit of public methods

Matching Invariants

```
public int ex1511(int[] b, int n)
{
    int s = 0;
    int i = 0;
    while (i != n) {
        s = s + b[i];
        i = i + 1;
    }
    return s;
}
```

A == B

Pattern

s size(b[])
sum(b[]) n
orig(n)
return ...

Variables

Matching Invariants

==	s	n	size (b[])	sum (b[])	orig (n)	ret
s		✗			✗	
n	✗			✗		✗
size(b[])						
sum(b[])		✗				
orig(n)	✗					✗
ret		✗			✗	

run 1

A == B

Pattern

s size(b[])
sum(b[]) n
orig(n)
return ...

Variables

Matching Invariants

==	s	n	size (b[])	sum (b[])	orig (n)	ret
s		✗	✗		✗	
n	✗			✗	✗	✗
size(b[])	✗			✗		✗
sum(b[])		✗	✗		✗	
orig(n)	✗	✗		✗		✗
ret		✗	✗		✗	

run 2

A == B

Pattern

s size(b[])
sum(b[]) n
orig(n)
return ...

Variables

Matching Invariants

==	s	n	size (b[])	sum (b[])	orig (n)	ret
s		✘	✘		✘	
n	✘			✘	✘	✘
size(b[])	✘			✘		✘
sum(b[])		✘	✘		✘	
orig(n)	✘	✘		✘		✘
ret		✘	✘		✘	

run 3

A == B

Pattern

s size(b[])
sum(b[]) n
orig(n)
return …

Variables

Matching Invariants

==	s	n	size (b[])	sum (b[])	orig (n)	ret
s		✘	✘		✘	
n	✘			✘	✘	✘
size(b[])	✘			✘		✘
sum(b[])		✘	✘		✘	
orig(n)	✘	✘		✘		✘
ret		✘	✘		✘	

s == sum(b[])

s == ret

n == size(b[])

ret == sum(b[])

Matching Invariants

```
public int ex1511(int[] b, int n)
{
    int s = 0;
    int i = 0;
    while (i != n) {
        s = s + b[i];
        i = i + 1;
    }
    return s;
}
```

s == sum(b[])

s == ret

n == size(b[])

ret == sum(b[])

Enhancing Relevance

- Handle polymorphic variables
- Check for derived values
- Eliminate redundant invariants
- Set statistical threshold for relevance
- Verify correctness with static analysis

polymorphic variables: treat “object x” like “int x” if possible
derived values: have “size(…)” as extra value to compare against
redundant invariants: like x > 0 => x >= 0
statistical threshold: to eliminate random occurrences
verify correctness: to make sure invariants **always** hold

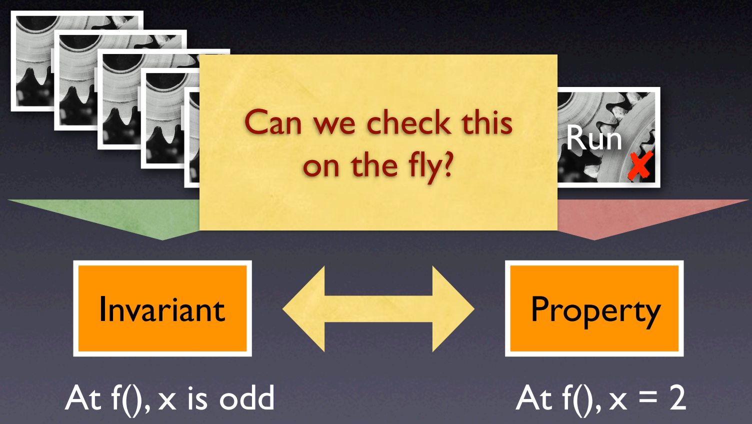
Daikon Discussed

- As long as some property can be observed, it can be added as a pattern
- Pattern vocabulary determines the invariants that can be found (“sum()”, etc.)
- Checking all patterns (and combinations!) is expensive
- Trivial invariants must be eliminated

Techniques

Dynamic Invariants	Value Ranges	Sampled Values

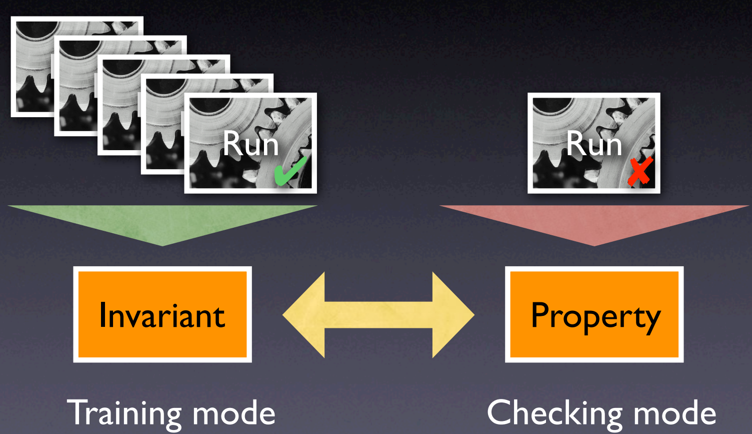
Dynamic Invariants



Diduce

- Determines *invariants* and *violations*
- Written by Sudheendra Hangal and Monica Lam (2001)
- Java bytecode
- analyzed > 30,000 lines of code

Diduce



Training Mode

- Start with empty set of invariants
- Adjust invariants according to values found during run

Invariants in Diduce

For each variable, Diduce has a pair (V, M)

- V = *initial value* of variable
- M = *range of values:* i-th bit of M is cleared if value change in i-th bit was observed
- With each assignment of a new value W, M is updated to M := M ∧ ¬ (W ⊗ V)
- *Differences* are stored in same format

Training Example

Code	i	Values		Differences		Invariant
		V	M	V	M	
`i = 10`	1010	1010	1111	–	–	i = 10
`i += 1`	1011	1010	1110	1	1111	10 ≤ i ≤ 11 ∧ |i′ − i| =
`i += 1`	1100	1010	1000	1	1111	8 ≤ i ≤ 15 ∧ |i′ − i| = 1
`i += 1`	1101	1010	1000	1	1111	8 ≤ i ≤ 15 ∧ |i′ − i| = 1
`i += 2`	1111	1010	1000	1	1101	8 ≤ i ≤ 15 ∧ |i′ − i| ≤

During *checking*, clearing an M-bit is an anomaly

Diduce vs. Daikon

- Less space and time requirements
- Invariants are computed on the fly
- Smaller set of invariants
- Less precise invariants

Techniques

Dynamic Invariants	Value Ranges	Sampled Values

Detecting Anomalies

Differences correlate with failure

Liblit's Sampling

- We want properties of runs in the field
- Collecting all this data is too expensive
- Would a *sample* suffice?
- Sampling experiment by Liblit et al. (2003)

Return Values

- Hypothesis: *function return values* correlate with failure or success
- Classified into positive / zero / negative

CCRYPT fails

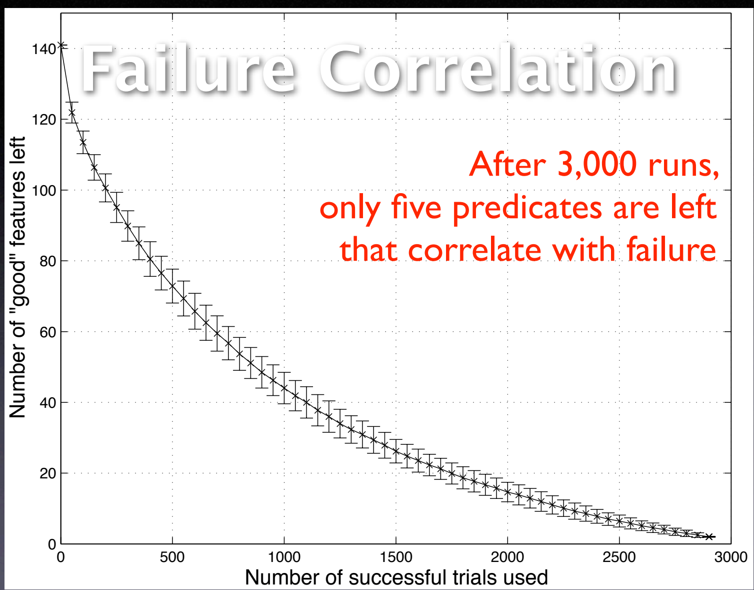
- CCRYPT is an interactive encryption tool
- When CCRYPT asks user for information before overwriting a file, and user responds with EOF, CCRYPT crashes
- 3,000 random runs
- Of 1,170 predicates, only file_exists() > 0 and xreadline() == 0 correlate with failure

Liblit's Sampling

Properties

- Can we apply this technique to remote runs, too?
- 1 out of 1000 return values was sampled
- Performance loss <4%

Failure Correlation



After 3,000 runs, only five predicates are left that correlate with failure

Web Services

- Sampling is first choice for web services
- Have 1 out of 100 users run an instrumented version of the web service
- Correlate instrumentation data with failure
- After sufficient number of runs, we can automatically identify the anomaly

Techniques

Dynamic Invariants	Value Ranges	Sampled Values

76

76

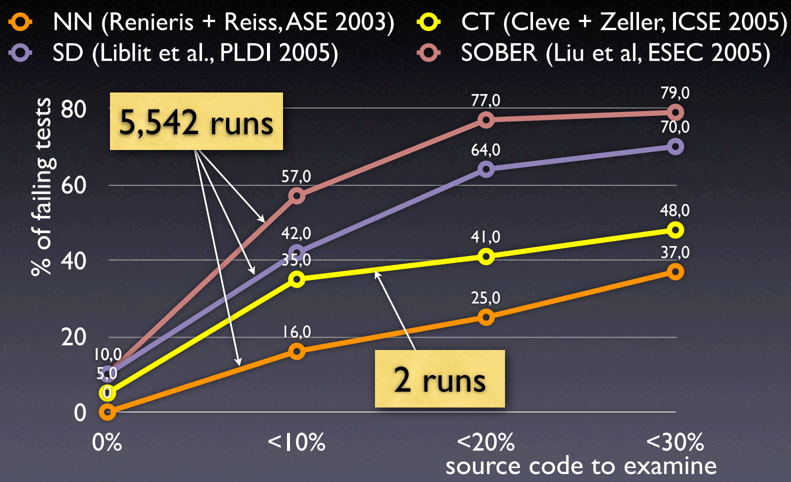
Anomalies and Causes

- An anomaly is not a cause, but a correlation
- Although correlation ≠ causation, anomalies can be excellent hints
- Future belongs to those who exploit
  - Correlations in *multiple runs*
  - Causation in *experiments*

77

77

Locating Defects



Results obtained from Siemens test suite; can not be generalized

78

78

NN (Nearest Neighbor) @Brown by Manos Renieris + Stephen Reiss
CT (Cause Transitions) @Saarland by Holger Cleve + Andreas Zeller
SD (Statistical Debugging) @Berkeley by Ben Liblit (now Wisconsin), Mayur Naik (Stanford), Alice Zheng, Alex Aiken (now Stanford), Michael Jordan
SOBER @Urbana-Champaign + Purdue by

Concepts

★ Comparing coverage (or other features) shows anomalies correlated with failure

★ Nearest neighbor or sequences locate errors more precisely than just coverage

★ Low overhead + simple to realize

Concepts (2)

★ Comparing data abstractions shows anomalies correlated with failure

★ Variety of abstractions and implementations

★ Anomalies can be excellent hints

★ Future: Integration of anomalies + causes