Detecting Anomalies
Andreas Zeller

Tracing Infections
• For every infection, we must find the earlier infection that causes it.
• Which origin should we focus upon?
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 ≤ x ≤ 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

```c
int main(int argc, 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;
}
```

```c
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;
}
```

$ middle 3 3 5
middle: 3

$ middle 2 1 3
middle: 1
Obtaining Coverage for C programs

```c
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

- executed only in failing runs: highly suspect
- executed in passing and failing runs: ambiguous
- executed only in passing runs: likely correct
```c
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;
}
```
Hue

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

0% passed 100% passed

Brightness

\[ \text{bright}(s) = \max(\%\text{passed}(s), \%\text{failed}(s)) \]

frequently executed

rarely executed

```c
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
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)
18 of 20 defects are correctly classified in the "reddest" portion of the code.

The "reddest" portion is at most 20% of the code.

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)
Nearest Neighbor

Compare with the single run that has the most similar coverage

Locating Defects

- Nearest Neighbor (Renieris+Reiss (ASE 2003))
- Intersection (Jones et al. (ICSE 2002))

Results obtained from Siemens test suite cannot be generalized
Sequences

Sequences of locations can correlate with failures:

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>open() read() close()</td>
<td>✔</td>
</tr>
<tr>
<td>open() close() read()</td>
<td>✘</td>
</tr>
<tr>
<td>close() open() read()</td>
<td>✘</td>
</tr>
</tbody>
</table>

...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:

\[
\text{ThisJoinPointVisitor.isRef()}, \quad \text{ThisJoinPointVisitor.canTreatAsStatic()}, \quad \text{MethodDeclaration.traverse()}, \quad \text{ThisJoinPointVisitor.isRef()}, \quad \text{ThisJoinPointVisitor.isRef()}
\]

Collecting Sequences

Ingoing vs. Outgoing
**Anomalies**

- **Weights**
  - Passing run
  - Passing run
  - Failing run

- **Ranking by average weight**
  - 0.60
  - 0.50
  - 0.40

**NanoXML**

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

**Locating Defects**

- AMPLEx/window size 8
- Dallmeier et al. (ECOOP 2005)

Results obtained from NanoXML can not be generalized
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()”

Techniques

Dynamic Invariants

Value Ranges

Sampled Values
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;
}

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

Daikon

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

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

Getting the Trace

• Records all variable values at all function entries and exits
• Uses VALGRIND to create the trace
Filtering Invariants

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

Method Specifications

using *primitive data*

<table>
<thead>
<tr>
<th>x = 6</th>
<th>x ∈ {2, 5, −30}</th>
<th>x &lt; y</th>
</tr>
</thead>
<tbody>
<tr>
<td>y = 5x + 10</td>
<td>z = 4x + 12y + 3</td>
<td>z = fn(x, y)</td>
</tr>
</tbody>
</table>

using *composite data*

<table>
<thead>
<tr>
<th>A subseq B</th>
<th>x ∈ A</th>
<th>sorted(A)</th>
</tr>
</thead>
</table>

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
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;
}
Matching Invariants

<table>
<thead>
<tr>
<th>==</th>
<th>s</th>
<th>n</th>
<th>size(b[])</th>
<th>sum(b[])</th>
<th>orig(n)</th>
<th>ret</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>n</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>size(b[])</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>sum(b[])</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>orig(n)</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>ret</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>

Pattern

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

Variables

run 3

Matching Invariants

<table>
<thead>
<tr>
<th>==</th>
<th>s</th>
<th>n</th>
<th>size(b[])</th>
<th>sum(b[])</th>
<th>orig(n)</th>
<th>ret</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>n</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>size(b[])</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>sum(b[])</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>orig(n)</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>ret</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>

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;
}
Enhancing Relevance

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

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

<table>
<thead>
<tr>
<th>Dynamic Invariants</th>
<th>Value Ranges</th>
<th>Sampled Values</th>
</tr>
</thead>
</table>

polymorphic variables: treat “object x” like “int x” if possible
derived values: have “size (...)” as extra value to compare against
redundant invariants: like \( x > 0 \Rightarrow x \geq 0 \)
statistical threshold: to eliminate random occurrences
verify correctness: to make sure invariants \textbf{always} hold
Dynamic Invariants

Can we check this on the fly?

Invariant
At f(), x is odd

Property
At f(), x = 2

Diduce

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

Diduce

Invariant
Training mode

Property
Checking mode
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 $M := M \land \neg (W \oplus V)$
- Differences are stored in same format

Training Example

<table>
<thead>
<tr>
<th>Code</th>
<th>i</th>
<th>Values</th>
<th>Differences</th>
<th>Invariant</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>V</td>
<td>M</td>
<td></td>
</tr>
<tr>
<td>i = 10</td>
<td>1010</td>
<td>1010</td>
<td>1111</td>
<td>i = 10</td>
</tr>
<tr>
<td>i += 1</td>
<td>1011</td>
<td>1010</td>
<td>1110</td>
<td>10 ≤ i ≤ 11 ∧</td>
</tr>
<tr>
<td>i += 1</td>
<td>1100</td>
<td>1010</td>
<td>1000</td>
<td>8 ≤ i ≤ 15 ∧</td>
</tr>
<tr>
<td>i += 1</td>
<td>1101</td>
<td>1010</td>
<td>1000</td>
<td>8 ≤ i ≤ 15 ∧</td>
</tr>
<tr>
<td>i += 2</td>
<td>1111</td>
<td>1010</td>
<td>1000</td>
<td>8 ≤ i ≤ 15 ∧</td>
</tr>
</tbody>
</table>

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

How do we collect data in the field?

Properties

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

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

Properties

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

<table>
<thead>
<tr>
<th>Dynamic Invariants</th>
<th>Value Ranges</th>
<th>Sampled Values</th>
</tr>
</thead>
</table>

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

Locating Defects

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