Comparing Coverage

Andreas Zeller
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
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 \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

$ \textit{middle} \ 3 \ 3 \ 5
\textit{middle:} \ 3

$ \textit{middle} \ 2 \ 1 \ 3
\textit{middle:} \ 1
```c
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
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
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 + "(0):", \ 
    event, arg
    return tracer
Obtaining Coverage for Python programs

$ ./middle.py 3 3 5

For remaining steps, see new project
```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;
}
```
```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;
}
```
Continuous Coloring

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

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

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

Results obtained from Siemens test suite; can not be generalized.
Sequences

Sequences of locations can correlate with failures:

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

…but all locations are executed in both runs!
$ 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/
...
$

The AspectJ Compiler
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:

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

Defect location
Collecting Sequences

Trace

anInputStreamObj
mark read read skip read read skip read
mark read
read read
read skip
skip read
mark read
read read
read skip
skip read

Sequences

InputStream

Sequence Set
mark read
skip read
read read
read skip
read skip
skip read
Ingoing vs. Outgoing

- **aProducer**
  - add
  - isEmpty
  - get
  - isEmpty
  - add
  - add
  - incoming calls

- **aConsumer**
  - add
  - isEmpty
  - size
  - get
  - isEmpty
  - add
  - add
  - outgoing calls

- **aQueue**
  - add
  - isEmpty
  - size
  - get
  - removeFirst
  - firstElement

- **aLinkedList**
  - add
  - size
  - add
  - add

- **aLogger**
  - add
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

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

% of failing tests

on average 0.5 classes
less than window size 1

Classes to examine (of 16)
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