Comparing Coverage
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

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

$ 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

if __name__ == "__main__":
    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

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

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 + "()\n    event, arg
    return tracer
Getting 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:17:middle(): line None
./middle.py:18:middle(): line None
./middle.py:19:middle(): line None
./middle.py:26:middle(): line None
./middle.py:27:middle(): return 3
middle: 3
```

For remaining steps, see new project

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

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

Source: Jones et al., ICSE 2002

```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 = x;
        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

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

Results obtained from Siemens test suite cannot be generalized

% of failing tests

% of executed source code to examine
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

This work is licensed under the Creative Commons Attribution License. To view a copy of this license, visit http://creativecommons.org/licenses/by/1.0 or send a letter to Creative Commons, 1598 Indiana Ave, Stanford, California 94305, USA.