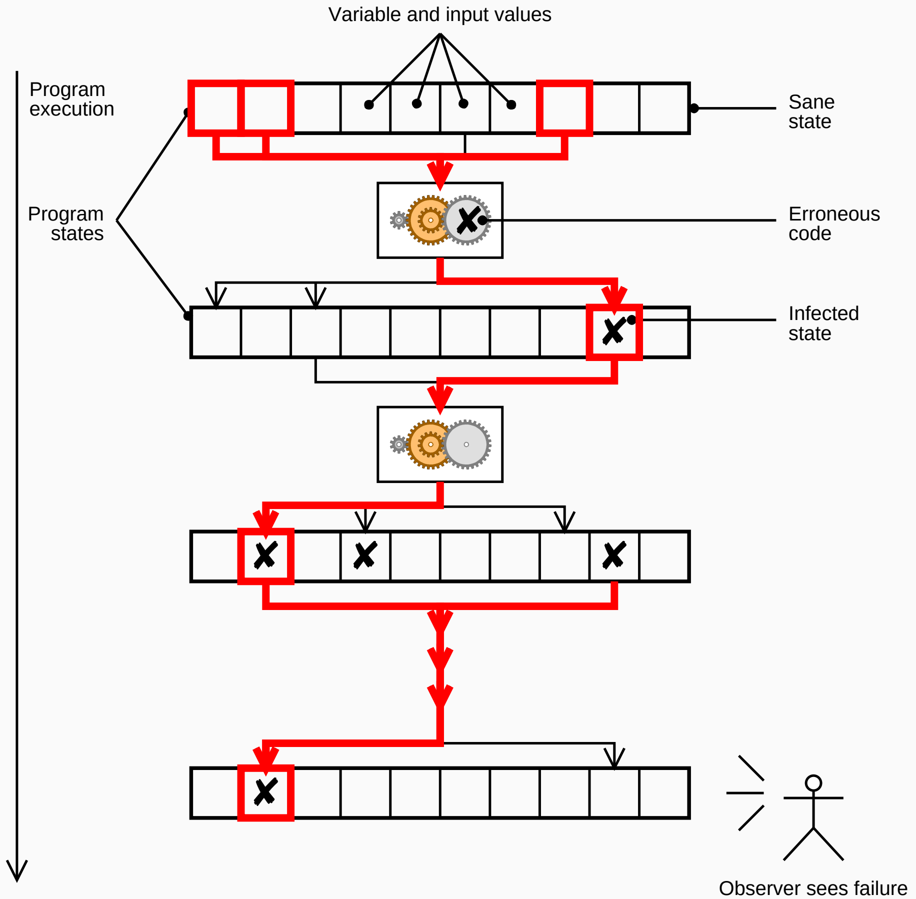
Detecting Anomalies

Andreas Zeller

Lehrstuhl Softwaretechnik
Universität des Saarlandes, Saarbrücken

Cause-Effect Chains



Where to search?

We can easily isolate *infectuous data* (automatically) . . .

- Tracing Origins
- Cause-Effect Chains

. . . and we can isolate *the infections themselves* (manually)

- Querying Events and Data
- Assertions

But where do we *start* with the search?

Focusing on Anomalies

Basic idea:

Start search where something *abnormal* happens!

How do we know what's normal and what's abnormal?

We use *induction*—reasoning from the particular to the general:

- Start with a *multitude* of program runs (= testcases).
- Determine properties that are *common* to all (or most) runs.

What's abnormal?

Suppose we have a multitude of *passing* runs and determine their normal properties.

Now we examine runs which *fail* the test.

Obviously, any such run differs from a single passing run at many places—just like any two runs differ.

However, focusing on the properties that deviate from *all* normal runs is likely to show failure causes.

Detecting Anomalies in a Nutshell

1. Separate runs into passing and failing
2. Determine *normal* properties of all passing runs.
3. For any failing run, find out which properties *deviate* from these normal properties.
4. Focus on these in the later search.

We'll explore two techniques:

- Detecting anomalies in *covered statements*
- Detecting anomalies in *invariants*

Detecting Coverage

Basic issue: *A good test suite should execute all statements* of the executed program.

(This is a *minimal criterion*; a better criterion would be to execute all branches, or all loops at least once, or. . .)

Realized by *instrumenting* the code.

Example: middle.c


```
// Return the middle of x, y, z
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;
}
```

Example: middle.c (2)


```
// Test driver
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;
}
```

```
$ gcc -o middle middle.c
$ ./middle 3 3 5
middle: 3
$ ./middle 2 1 3
middle: 1
$ _
```

Instrumenting middle.c

The compiler (GCC) can insert special *instrumentation code* into the executable:

```
$ gcc -g -fprofile-arcs -ftest-coverage -o middle \
        middle.c
```

This results in

- extra `.bb` and `.bbg` files created at compile time. These files map basic blocks and transition arcs to source code.
- extra `.da` files being created at execution. This file *summarizes* the coverage of transition arcs in *all* runs.

Instrumenting middle.c (2)

Executing `middle` summarizes the runs in a `.da` file:

```
$ rm *.da
$ ls
middle middle.bb middle.bbg middle.c
$ ./middle 3 3 5
middle: 3
$ ./middle 2 1 3
middle: 1
$ ls
middle middle.bb middle.bbg middle.c middle.da
$ _
```

Obtaining Coverage

gcov maps the recorded coverage to source code:

```
$ gcov middle.c
76.19% of 21 source lines executed in file middle.c
Creating middle.c.gcov.
$ _
```

That's not enough coverage yet—let's see if we can improve:

```
$ ./middle 3 2 1
middle: 2
$ gcov middle.c
85.71% of 21 source lines executed in file middle.c
Creating middle.c.gcov.
$ _
```

The middle.c.gcov file

```
            int middle(int x, int y, int z)
     3          int m = z;
     3          if (y < z)
     2             if (x < y)
######                m = y;
     2             else if (x < z)
     2                m = y;
     2           else
     1              if (x > y)
     1                 m = y;
######              else if (x > z)
######                 m = x;
     3

     3          return m;
     3
```

######: Code that was *not* executed so far

Coverage Anomaly

● (green), ● (red): covered statements

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

Discrete Visualization

Basic idea: differentiate

- Code executed only in failing runs
  ⇒ “Code that is highly suspect”
- Code executed in passing and failing runs
  ⇒ “Code that is ambiguous”
- Code executed only in passing runs
  ⇒ “Code that is probably correct”

Discrete Visualization (2)

Highlight code according to coverage

	3 3 5	1 2 3	3 2 1	5 5 5	5 3 4	2 1 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 Visualization

Have *hue* express *percentage* of failed statements:

- 100% test cases executing this statement failed
- 66% test cases executing this statement failed
- 50% test cases executing this statement failed
- 0% test cases executing this statement failed

Have *brightness* express *percentage* of test cases:

- executed in 100% of all test cases
- executed in 66% of all test cases
- executed in 33% of all test cases

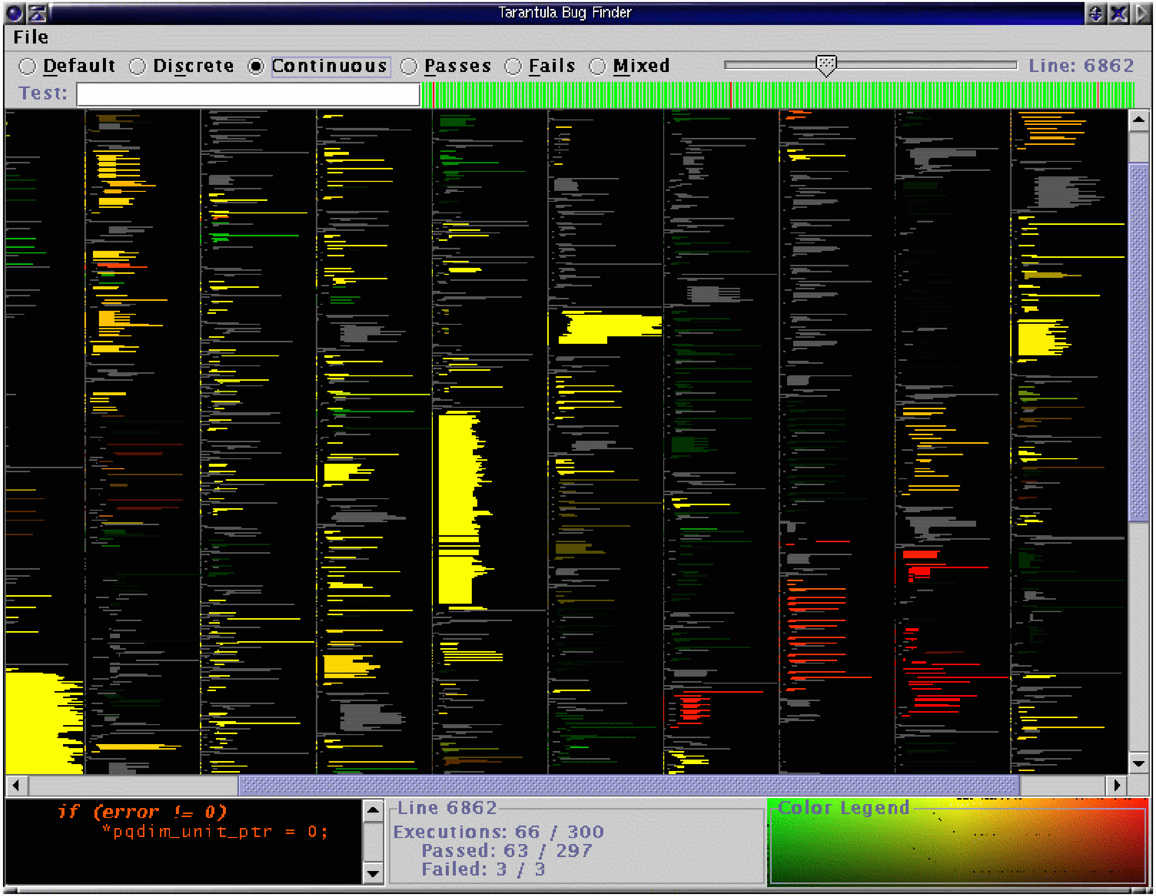
Continuous Visualization (2)

Highlight code according to coverage

```
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;
}
```

	3 3 5	1 2 3	3 2 1	5 5 5	5 3 4	2 1 3
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;	●	●	●	●	●	●
	✔	✔	✔	✔	✔	✘

The Tarantula Bug Finder



Significance of Coverage Anomaly

How well can comparing coverage detect anomalies?

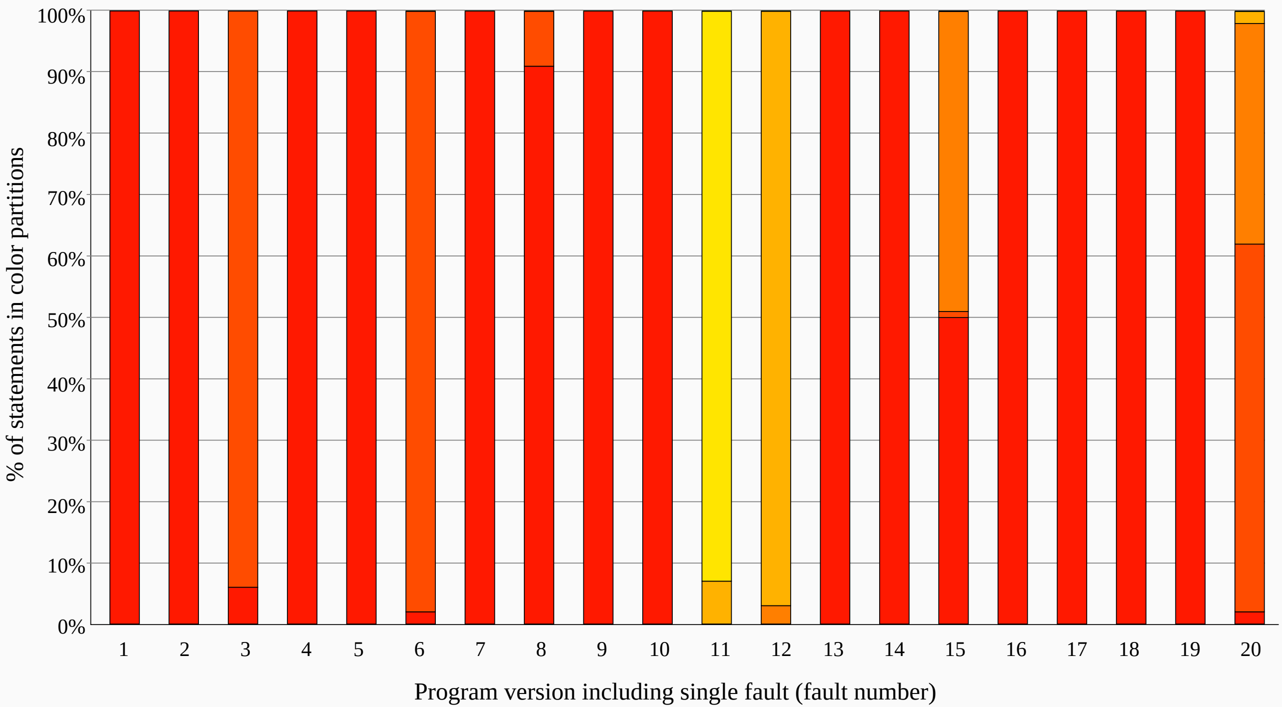
False negatives. "How red are the defective statements?"

False positives. "How red are non-defective statements?"

Subject program: *Space*

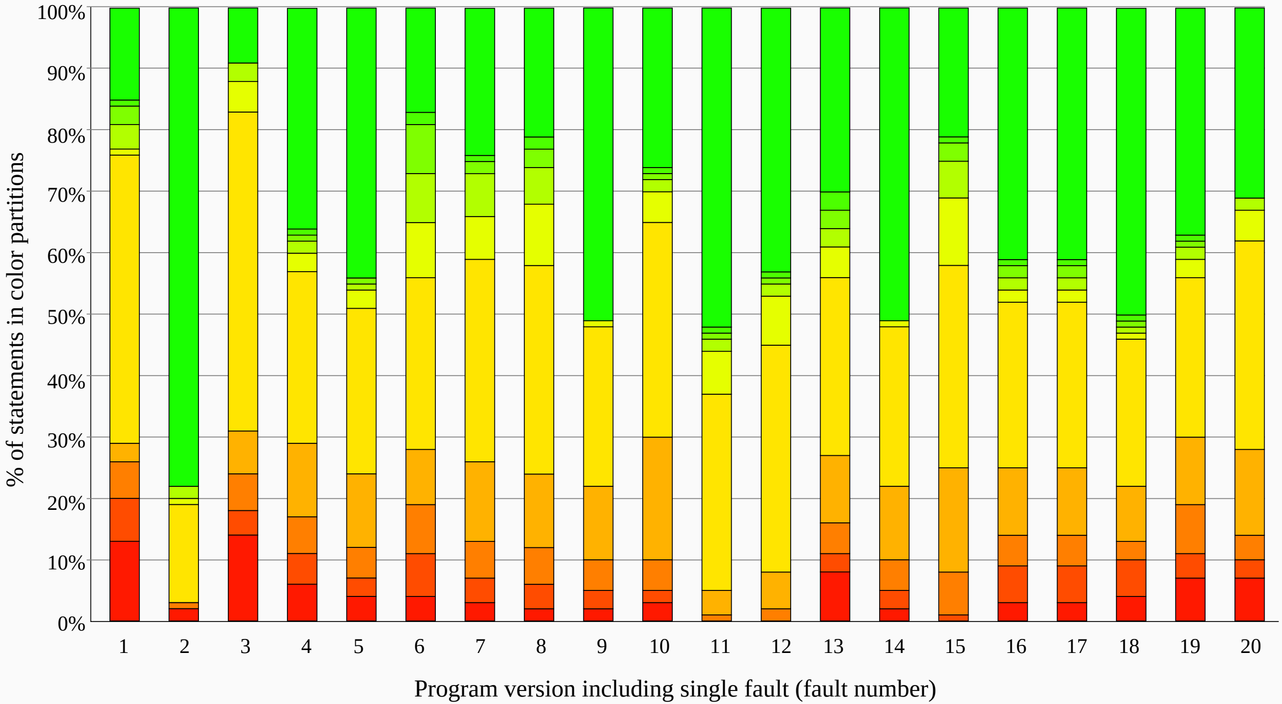
- 8000 lines of executable code
- 1000 coverage-based test suites w/ 156–4700 test cases
- 20 defective versions w/ one defect each
  (corrected in subsequent version)

How red are the defective statements?



Almost all defective statements were correctly flagged as red.

How red are non-defective statements?



Up to 10% of non-defective statements (≈ 800 statements) were incorrectly flagged as red.

Open Questions

- What is “the defective statement”? (The one that is to be changed?)
- What does it mean if a non-defective statement is red?
- What does it mean if a defective statement is not red?
- How many test runs are required?
- Can we generate extra test runs by experimentation?

Interaction with cause-effect chains

How can we integrate anomaly detection into isolation of origins and cause-effect chains?

Possible approach:

- Start with anomalies (i.e. red statements)
- Isolate origin of the statement being executed (i.e. the *if*-condition that is once false and once true)
- Test whether an alternate condition would have altered the outcome
- If outcome is altered, isolate the cause-effect chain of this condition, focusing on earlier anomalies

Open research (hint, hint)!

Concepts

- Comparing the coverage of passing and failing test cases can lead to statements likely to induce a failure.
- Technique is easy to use; results are easy to interpret
- Approach is based on *heuristics* (“likely to induce a failure”)
- Code that is *always executed* cannot be isolated (say, the `sample` defect)
- Approach depends on availability of large test suites

References

- *gcov—a test coverage program*, in: *Using the GNU Compiler*, http://gcc.gnu.org/onlinedocs/gcc/
- *Tarantula: Fault Localization via Visualization*, http://www.cc.gatech.edu/aristotle/Tools/tarantula/publications.html