Fault Localization

Overview

- Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique [1]
- Locating Causes of Program Failure [2]

Fault Localization

- Debugging software is an expensive and mostly manual process
- Of all debugging activities, locating the faults, or fault localization, is the most challenging one
- Approaches have been investigated to help automate fault localization

3

Typical Fault Localization Techniques

- Tarantula
- Set Union & Set Intersection
- Nearest Neighbor
- Cause Transitions

4

What Is the Fault in the Following Buggy Program?

```
int mid(int x, int y, int z) {
   int m;
   m= z;
   if (y < z) {
     if (x < y) m = y;
     else if (x < z) m = y;        // should be m = x;
} else {
       if (x > y) m = y;
       else if ( x > z) m = x;
     }
   } return m;
}
```


Tarantula: Coverage-based Fault Localization

Statements	3,3,5	1,2,3	3,2,1	5,5,5	5,3,4	2,1,3
`int m;`						
`m = z;`						
`if (y < z) {`						
`if (x < y)`						
`m = y;`						
`else if (x < z)`						
`m = y; //should be x`						
`} else {`						
`if (x > y)`						
`m = y;`						
`else if (x > z)`						
`m = x; }`						
`return m;`						
	Pass	Pass	Pass	Pass	Pass	Fail

Approach

- Insight
  - Entities in a program that are primarily executed by failed test cases are more likely to be faulty than those that are primarily executed by passed test cases
- Solution
  - Ranking based on suspiciousness

$$Suspicious(s) = \frac{fail(s)/totalfail}{fail(s)/totalfail + pass(s)/totalpass}$$

7

Tarantula

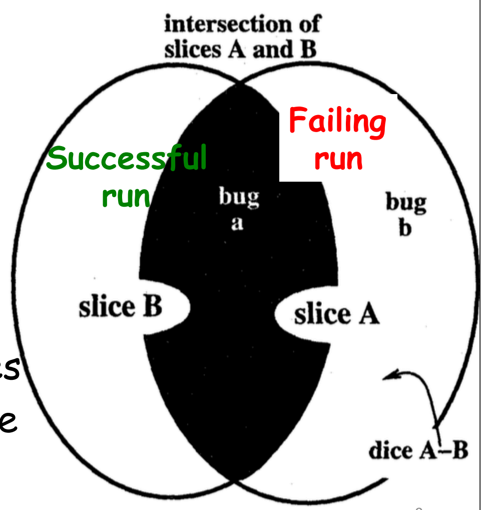
Statements	3,3,5	1,2,3	3,2,1	5,5,5	5,3,4	2,1,3	Susp
`int m;`							0.5
`m = z;`							0.5
`if (y < z) {`							0.5
`if (x < y)`							0.63
`m = y;`							0
`else if (x < z)`							0.71
`m = y; //should be x`							0.83
`} else {`							0
`if (x > y)`							0
`m = y;`							0
`else if (x > z)`							0
`m = x; }`							0
`return m;`							0.5
	Pass	Pass	Pass	Pass	Pass	Fail	8

(1/1)/(1/1+5/5)

(1/1)/(1/1+1/5)

Set Union & Set Intersection [3]

- Slice-based Fault Localization
  - A dynamic slice is the set of statements which ***do affect*** the value of the output
  - Dice: the set difference of two slices
  - dice (A – B) is effective to isolate bug b



9

Formulas

- Set Union

$$E_{initial} = E_f - \bigcup_{p \in P} E_p$$

- Set Intersection

$$E_{initial} = \bigcap_{p \in P} E_p - E_f$$

- What is the insight behind each formula?

10

Set Union & Set Intersection

	Statements	3,3,5	1,2,3	3,2,1	5,5,5	5,3,4	2,1,3
1	`int m;`	■	■	■	■	■	■
2	`m = z;`	■	■	■	■	■	■
3	`if (y < z) {`	■	■	■	■	■	■
4	`if (x < y)`	■	■			■	■
5	`m = y;`		■				
6	`else if (x < z)`	■				■	■
7	`m = y; //should be x`	■					■
8	`} else {`			■	■		
9	`if (x > y)`			■	■		
10	`m = y;`			■			
11	`else if (x > z)`				■		
12	`m = x; }`						
13	`return m;`	■	■	■	■	■	■
		Pass	Pass	Pass	Pass	Pass	Fail

Nothing is found!

11

Nearest Neighbor [4]

- Spectra-based Fault Localization
  - Spectrum: profiling data that shows the number of times each program line is executed
  - Given a set of passing tests and a failing test F, find the passing test P, which has the most similar spectrum as F
  - Calculate the distance metric

12

Two Variants

- NN/perm
  - Frequency-marked statements
  - Sort statements based on frequency
  - Ulam edit distance
    - E.g., Dist([a, b, c, d], [a, c, d, b]) = 1 (move)
- NN/binary
  - 0-or-1 mark for each statement
  - No frequency is considered
  - Set subtraction is used to calculate distance

13

Nearest Neighbor

	Statements	3,3,5	1,2,3	3,2,1	5,5,5	5,3,4	2,1,3
1	`int m;`	■	■	■	■	■	■
2	`m = z;`	■	■	■	■	■	■
3	`if (y < z) {`	■	■	■	■	■	■
4	`if (x < y)`	■	■			■	■
5	`m = y;`		■				
6	`else if (x < z)`	■				■	■
7	`m = y; //should be x`	■					■
8	`} else {`			■	■		
9	`if (x > y)`			■	■		
10	`m = y;`			■			
11	`else if (x > z)`				■		
12	`m = x; }`						
13	`return m;`	■	■	■	■	■	■
		Pass	Pass	Pass	Pass	Pass	Fail

Nothing is found!

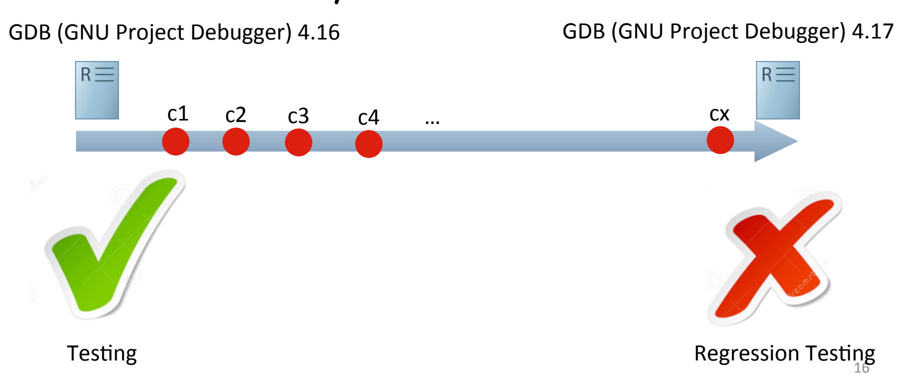
14

Cause-Transitions [2]

- Leverage delta debugging to isolate failure-inducing variable values at specific program locations
- Identify the transition points between different failure-inducing variable values
- Consider the transition points as bug locations

Delta Debugging (DD) [5]

- Problem Statement
  - Yesterday, my program worked. Today, it does not. Why?



Definitions

- Configuration: the set of all applied changes $C = \{\Delta_1, \Delta_2, ..., \Delta_n\}$
  - $c \subseteq C$ represents a subset of changes
- Test: the function c → {✗, ✓, ?} to determine whether a configuration c leads to failure, success, or unresolved outcome of regression testing

17

How to Find the Minimum Failing-Inducing Changes?

- Naïve approach
  - Brute-force search: too expensive
- Efficient approach
  - Delta debugging: Binary search

18

Insight

- By finding the minimum set of changes whose application fails the test, Delta Debugging identify bug-inducing changes

Search for Single Failure-Inducing Change

- Suppose there are 8 changes with the 7^{th} is the cause. How do you use binary search to find it?

Conceptual Solution

Step	Configuration	test
1	1 2 3 4	✔
2	5 6 7 8	✗
3	5 6	✔
4	7 8	✗
5	7	✗

21

How Does DD Localize Failure-Inducing Variable Values?

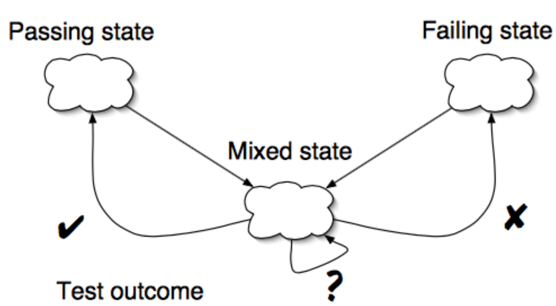


Figure 2: Narrowing down state differences. By assessing whether a mixed state results in a passing (✔), a failing (✘), or an unresolved (?) outcome, Delta Debugging isolates a relevant difference.

22

Cause-Transitions

	Statements	3,3,5	1,2,3	3,2,1	5,5,5	5,3,4	2,1,3
1	`int m;`						
2	`m = z;`						
3	`if (y < z) {`						
4	`if (x < y)`						
5	`m = y;`						
6	`else if (x < z)`						
7	`m = y; //should be x`						
8	`} else {`						
9	`if (x > y)`						
10	`m = y;`						
11	`else if (x > z)`						
12	`m = x; }`						
13	`return m;`						
		Pass					Fail

x, y, z

Step 1: Line 1: 3, 3, 3 ✔
2, 1, 5 ✗
2, 3, 5 ✔
3, 1, 5 ✗

x, y, z, m

Step 2: Line 13: 3, 3, 3, 1 ✗
3, 3, 3, 3 ✔
3, 3, 5, 1 ✗

Step 4: Line 4: 3, 3, 3, 3 ✔
2, 1, 5, 5 ✗
2, 3, 5, 5 ✔
3, 1, 5, 5 ✗

Step 5: Line 6: 3, 3, 3, 3 ✔
2, 1, 5, 5 ✗

23

Cause-Transitions

	Statements	3,3,5	1,2,3	3,2,1	5,5,5	5,3,4	2,1,3
1	`int m;`						
2	`m = z;`						
3	`if (y < z) {`						
4	`if (x < y)`						
5	`m = y;`						
6	`else if (x < z)`						
7	`m = y; //should be x`						
8	`} else {`						
9	`if (x > y)`						
10	`m = y;`						
11	`else if (x > z)`						
12	`m = x; }`						
13	`return m;`						
		Pass					Fail

2, 3, 5, 5 ✔
3, 1, 5, 5 ✗

Step 6: Line 7: 3, 3, 3, 1 ✗
3, 3, 3, 3 ✔
3, 3, 5, 1 ✗

Line 7 is the fault location!

24

Evaluation

- Siemens suite
  - 7 programs, 132 fault versions, 21,631 test suites designed to expose the faults
  - 122 versions are usable by the authors
  - Each version contains exactly one fault
  - Each fault may span multiple statements or even functions

Evaluation Method

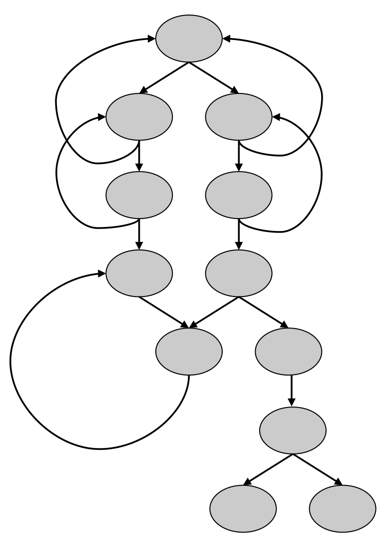
- Basic Idea
  - Imagine an “ideal” debugger or a perfect programmer examines the ranked list of bug locations
  - The fewer locations/statements examined before the actual location, the higher score the report/tool gets
    - Tarantula: go through the ranked list
    - Other tools: PDG-based location examination

PDG-Based Evaluation Method [6]

- Given a reported location, do breadth-first search of Program Dependency Graph (PDG)
  - Terminate the search when a real fault is found
  - Score is proportional to the unexplored part of the PDG
    - Score near 1.0 means the No. 1 reported location is the correct one.

27

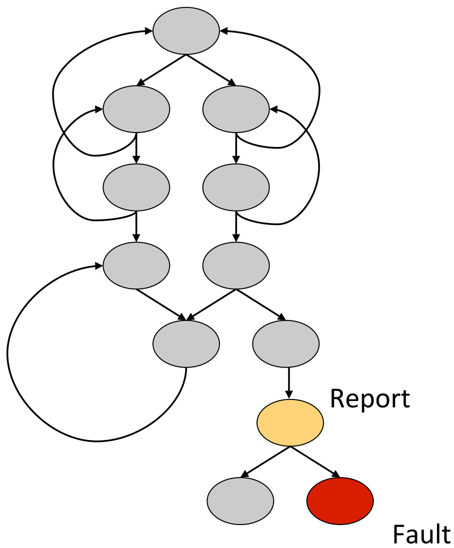
An Example [6]



12 total nodes in PDG

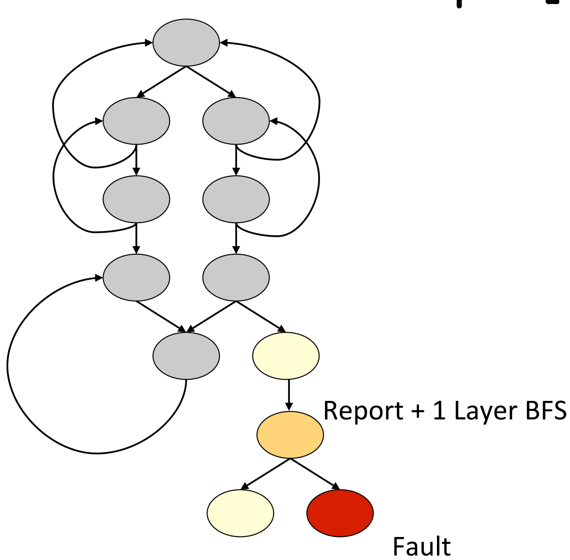
28

An Example [6]

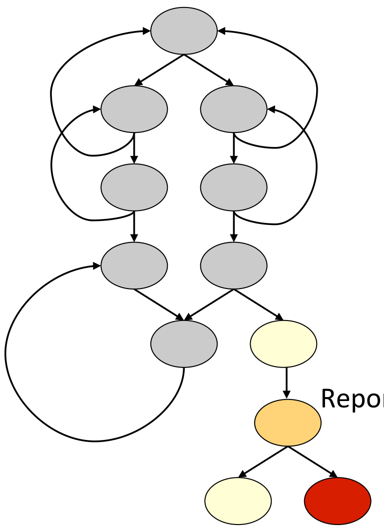


12 total nodes in PDG

An Example [6]



An Example [6]



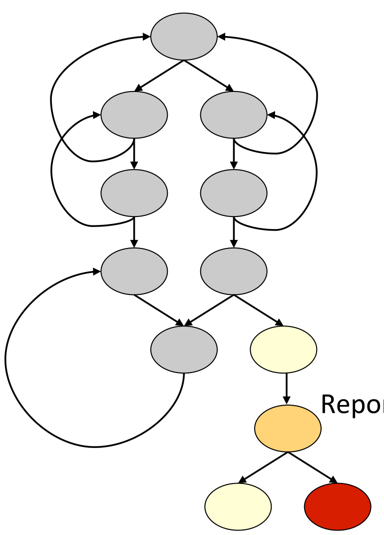
12 total nodes in PDG

Report + 1 Layer BFS

STOP: Real fault discovered

31

An Example [6]



12 total nodes in PDG
8 of 12 nodes not covered by BFS: score = 8/12 ~= 0.67.

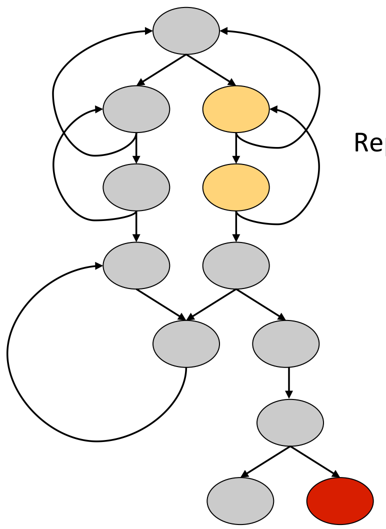
Report + 1 Layer BFS

STOP: Real fault discovered

Fault

32

An Example [6]



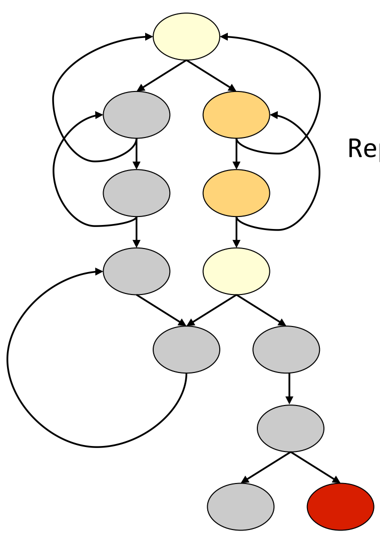
Report

12 total nodes in PDG

Fault

33

An Example [6]



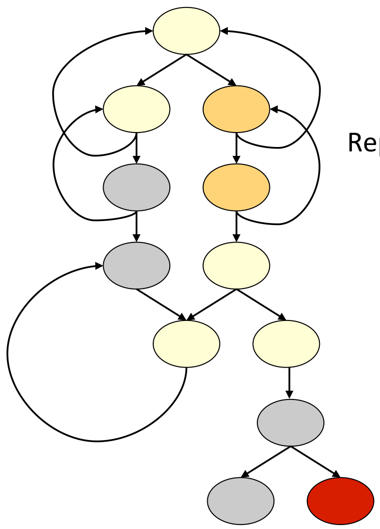
Report + 1 layer BFS

12 total nodes in PDG

Fault

34

An Example [6]

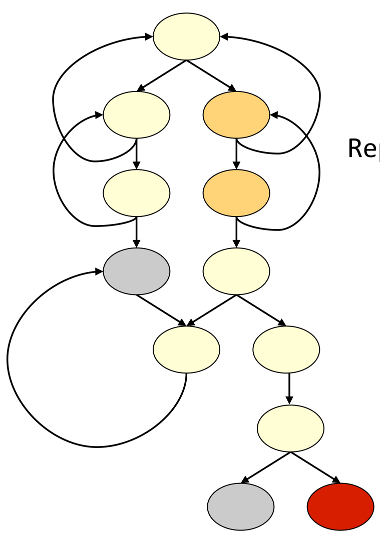


Report + 2 layers BFS

12 total nodes in PDG

Fault

An Example [6]

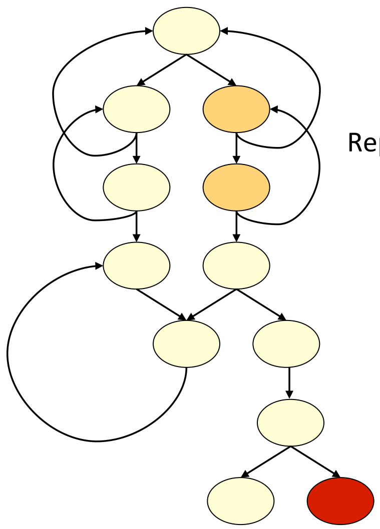


Report + 3 layers BFS

12 total nodes in PDG

Fault

An Example [6]



Report + 4 layers BFS

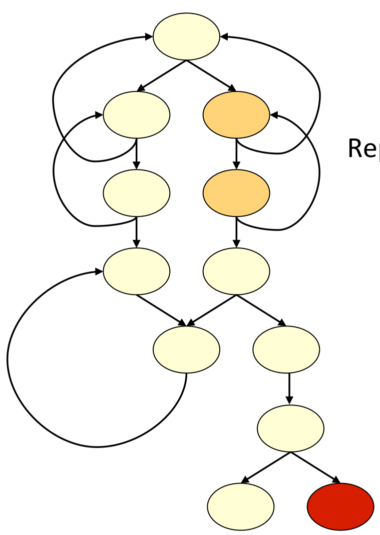
STOP: Real fault discovered

12 total nodes in PDG

Fault

37

An Example [6]



Report + 4 layers BFS

12 total nodes in PDG
0 of 12 nodes not covered by
BFS: score = 0/12 ~= 0.00.

38

Limitations [6]

- Isn't a misleading report worse than an empty report?
- Nobody really searches a PDG like that!

Evaluation Results

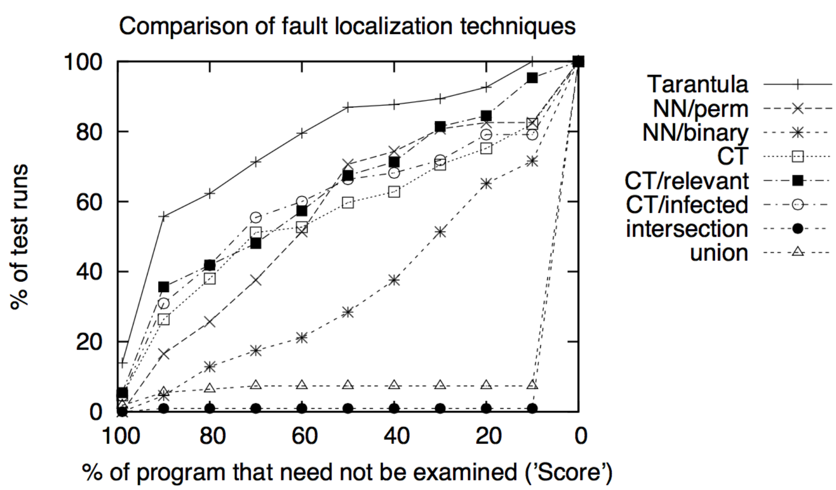


Figure 2: Comparison of the effectiveness of each technique.

Evaluation Results

Table 3: Average time expressed in seconds.

Program	Tarantula (computation only)	Tarantula (including I/O)	Cause Transitions
print_tokens	0.0040	68.96	2590.1
print_tokens2	0.0037	50.50	6556.5
replace	0.0063	75.90	3588.9
schedule	0.0032	30.07	1909.3
schedule2	0.0030	30.02	7741.2
tcas	0.0025	12.37	184.8
tot_info	0.0031	8.51	521.4

Reference

[1] J. A. Jones, and M. J. Harrold, Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique, ASE '05

[2] H. Cleve, and A. Zeller, Locating Causes of Program Failures, ICSE '05

[3] H. Agrawal, J. Horgan, S. London, and W. Wong. Fault Localization Using Execution Slices and Dataflow Tests, SRE '95

[4] M. Renieris and S. Reiss. Fault localization with nearest neighbor queries. ASE '03

[5] A. Zeller. Yesterday, My Program Worked. Today, It Does Not. Why?, FSE '99

[6] A. D. Groce, Testing and Debugging: Causality and Fault Localization, http://www.cs.cmu.edu/~agroce/CS119/l6.ppt.