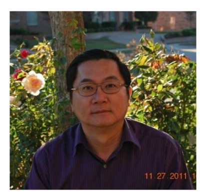
STAR Laboratory of Advanced Research on Software Technology

Reducing the Cost of Program Debugging with Effective Software Fault Localization

W. Eric Wong
Department of Computer Science
The University of Texas at Dallas
ewong@utdallas.edu
http://www.utdallas.edu/~ewong

Speaker Biographical Sketch



- Professor & Director of International Outreach
  Department of Computer Science
  University of Texas at Dallas
- Guest Researcher
  Computer Security Division
  National Institute of Standards and Technology (NIST)
- Vice President, IEEE Reliability Society
- Secretary, ACM SIGAPP (Special Interest Group on Applied Computing)
- Principal Investigator, NSF TUES (Transforming Undergraduate Education in Science, Technology, Engineering and Mathematics) Project
  - *Incorporating Software Testing into Multiple Computer Science and Software Engineering Undergraduate Courses*
- Founder & Steering Committee co-Chair for the SERE conference (*IEEE International Conference on Software Security and Reliability*) (http://paris.utdallas.edu/sere13)

Outline

- Motivation and Background
- Execution Dice-based Fault Localization
- Suspiciousness Ranking-based Fault Localization
  - Program Spectra-based Fault Localization
  - Code Coverage-based Fault Localization
  - Statistical Analysis-based Fault Localization
  - Neural Network-based Fault Localization
  - Similarity Coefficient-based Fault Localization
- Empirical Evaluation
- Theoretical Comparison: Equivalence
- Mutation-based Automatic Bug Fixing
- Conclusions

Motivation

- Testing and debugging activities constitute one of the most expensive aspects of software development
  - Often more than 50% of the cost [Hailpern & Santhanam, 2003]
- Manual debugging is…
  - Tedious
  - Time Consuming
  - Error prone
  - Prohibitively expensive

Need ways to debug…

automatically

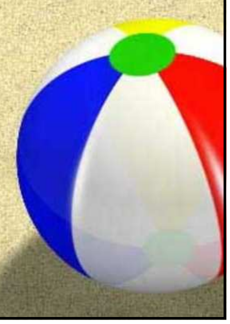
B. Hailpern and P. Santhanam, “Software Debugging, Testing, and Verification,” *IBM Systems Journal*, 41(1):4-12, 2002

Debugging Today

- Program debugging consists of three fundamental activities
  - Learning that the program has a fault – *fault detection*
  - Finding the location of the fault – *fault localization*
  - Actually removing the fault – *fault fixing*
- A lot of progress has been made in the area of test case generation and thus we can assume that we will have a collection of test cases (i.e., a test set) that can reveal that the program has faults.
  - So the programmer can avoid the first task (fault detection).
- Recently fault localization has received a lot of focus
  - It is one of the most expensive debugging activities [Vessey, 1985]
- Fault Fixing has also been an important research area
  - Have to be very careful not to introduce new faults in the process

Iris Vessy, "Expertise in Debugging Computer Programs: A Process Analysis," *International Journal of Man-Machine Studies*, 23(5):459-494, March1985

Software Fault Localization



Objectives

- Develop a robust and reliable fault localization technique to identify faults from *dynamic behaviors* of programs
- Reduce the cost of program debugging by providing *a more accurate set of candidate fault positions*
- Provide software engineers with effective tool support

Perfect Bug Detection

- A bug in a statement will be detected by a programmer if the statement is examined
  - A correct statement will not be mistakenly identified as a faulty statement
  - If the assumption does not hold, a programmer may need to examine more code than necessary in order to find a faulty statement

Traditional Approach

Commonly Used Techniques

- Insert *print* statements
- Add *assertions* or set *breakpoints*
- Examine core dump or stack trace

Rely on programmers' intuition and domain expert knowledge

Outline

- Motivation and Background
- Execution Dice-based Fault Localization
- Suspiciousness Ranking-based Fault Localization
  - Program Spectra-based Fault Localization
  - Code Coverage-based Fault Localization
  - Statistical Analysis-based Fault Localization
  - Neural Network-based Fault Localization
  - Similarity Coefficient-based Fault Localization
- Empirical Evaluation
- Theoretical Comparison: Equivalence
- Mutation-based Automatic Bug Fixing
- Conclusions

Execution Dice-based Fault Localization

Execution Slice & Dice

- Faults reside in the execution slice of a test that fails on execution
  - An execution slice is the set of a program's code (blocks, statements, decisions, c-uses, or p-uses) executed by a test
  - An execution slice can be constructed very easily if we know the coverage of the test (instead of reporting the coverage percentage, it reports which parts of the program are covered).
  - Too much code in the slice
- Narrowing search domain by execution dices
  - An execution dice is obtained by subtracting *successful* execution slices from *failed* execution slices

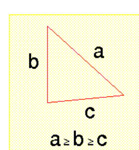
Dice = Execution slices of failed tests – Execution slices of successful tests

Static & Dynamic | Discussion 1 | Discussion 2

Example (1)

A Sample Program

```
read (a, b, c);
class = scalene;
if a = b || b = a
    class = isosceles;
if a*a = b*b + c*c
    class = right;
if a = b && b = c
    class = equilateral;
case class of
    right        :  area = b*c / 2;
    equilateral  :  area = a*a * sqrt(3)/4;
    otherwise    :  s = (a+b+c)/2;
                    area = sqrt(s*(s-a)*(s-b)*(s-c));
end;
write(class, area);
```



Example (2)

Initial Test Set

Test case	Input			Output	
	a	b	c	class	area
T_1	2	2	2	equilateral	1.73
T_2	4	4	3	isosceles	5.56
T_3	5	4	3	right	6.00
T_4	6	5	4	scalene	9.92
T_5	3	3	3	equilateral	3.90

Example (3)

Failure Detected

Test case	Input			Output	
	a	b	c	class	area
T_1	2	2	2	equilateral	1.73
T_2	4	4	3	isosceles	5.56
T_3	5	4	3	right	6.00
T_4	6	5	4	scalene	9.92
T_5	3	3	3	equilateral	3.90
T_6	4	3	3	scalene	4.47

Failure!

Example (4)

Where is the Bug?

```
read (a, b, c);        4, 3, 3
class = scalene;
if a = b || b = a
    class = isosceles;
if a*a = b*b + c*c
    class = right;
if a = b && b = c
    class = equilateral;
case class of
    right        :  area = b*c / 2;
    equilateral  :  area = a*a * sqrt(3)/4;
    otherwise    :  s = (a+b+c)/2;
                    area = sqrt(s*(s-a)*(s-b)*(s-c));
end;
write(class, area);        scalene
```


Example (5)

Execution Slice w.r.t. the Failed Test T_6 = (4 3 3)

```
read (a, b, c);
class = scalene;
if a = b || b = a
    class = isosceles;
if a*a = b*b + c*c
    class = right;
if a = b && b = c
    class = equilateral;
case class of
    right        :  area = b*c / 2;
    equilateral  :  area = a*a * sqrt(3)/4;
    otherwise    :  s = (a+b+c)/2;
                    area = sqrt(s*(s-a)*(s-b)*(s-c));
end;
write(class, area);
```

Too much code needs
To be examined!

Example (6): Which Test Should be Used?

Failure Detected

Test case	Input			Output	
	a	b	c	class	area
T_1	2	2	2	equilateral	1.73
T_2	4	4	3	isosceles	5.56
T_3	5	4	3	right	6.00
T_4	6	5	4	scalene	9.92
T_5	3	3	3	equilateral	3.90
T_6	4	3	3	scalene	4.47

Failure!

Example (7)

A Successful Test T_2 and a Failed Test T_6

Test case	Input			Output	
	a	b	c	class	area
T_1	2	2	2	equilateral	1.73
T_2	4	4	3	isosceles	5.56
T_3	5	4	3	right	6.00
T_4	6	5	4	scalene	9.92
T_5	3	3	3	equilateral	3.90
T_6	4	3	3	scalene	4.47

Success

Failure! (should be ***isosceles***)

Example (8)

Execution Slice w.r.t. the Successful Test T_2 = (4 4 3)

```
read (a, b, c);
class = scalene;
if a = b || b = a
    class = isosceles;
if a*a = b*b + c*c
    class = right;
if a = b && b = c
    class = equilateral;
case class of
    right       :  area = b*c / 2;
    equilateral :  area = a*a * sqrt(3)/4;
    otherwise   :  s = (a+b+c)/2;
                   area = sqrt(s*(s-a)*(s-b)*(s-c));
end;
write(class, area);
```


Example (9)

Execution Dice = Slice (4 3 3) - Slice (4 4 3)

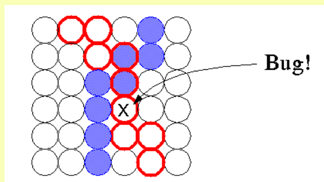
```
read (a, b, c);
class = scalene;
if a = b || b = a
    class = isosceles;
if a*a = b*b + c*c
    class = right;
if a = b && b = c
    class = equilateral;
case class of
    right       :  area = b*c / 2;
    equilateral :  area = a*a * sqrt(3)/4;
    otherwise   :  s = (a+b+c)/2;
                   area = sqrt(s*(s-a)*(s-b)*(s-c));
end;
write(class, area);
```

Bug! Should be b=c

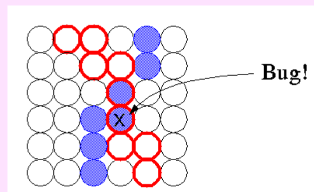
One Failed and One Successful Test

Possible locations of faults

- Code in the execution dice (top priority)
- A bug is in the failed execution slice (the red path) but not in the successful execution slice (the blue path)



- Code in the failed execution slice but not in the dice
- A bug is in the failed execution slice (the red path) and in the successful execution slice (the blue path)

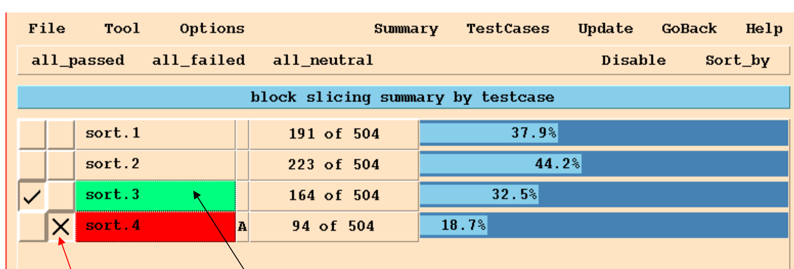


- The dicing-based technique can be effective in locating some program bugs
  - H. Agrawal, J. R. Horgan, S. London, and W. E. Wong, "Fault localization using execution slices and dataflow tests," *in Proceedings of the 6th IEEE International Symposium on Software Reliability Engineering*, pp. 143-151, Toulouse, France, October 1995.

    †Authors are listed in alphabetical order

    ‡Number of citations: 155 (according to the Google Scholar)

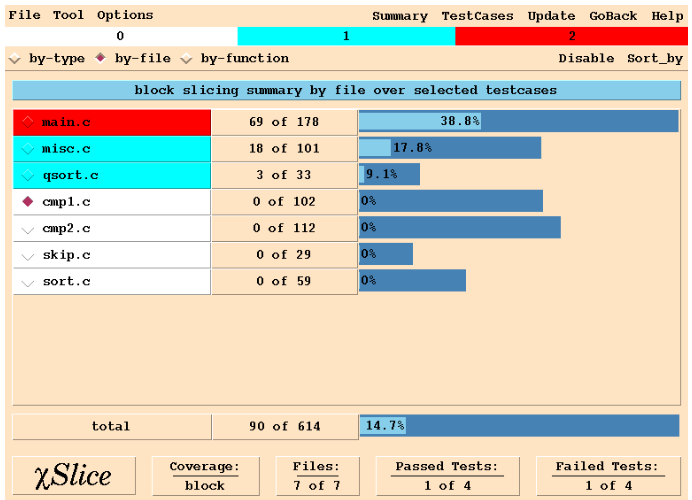
Locating Bugs using Execution Dice (1)



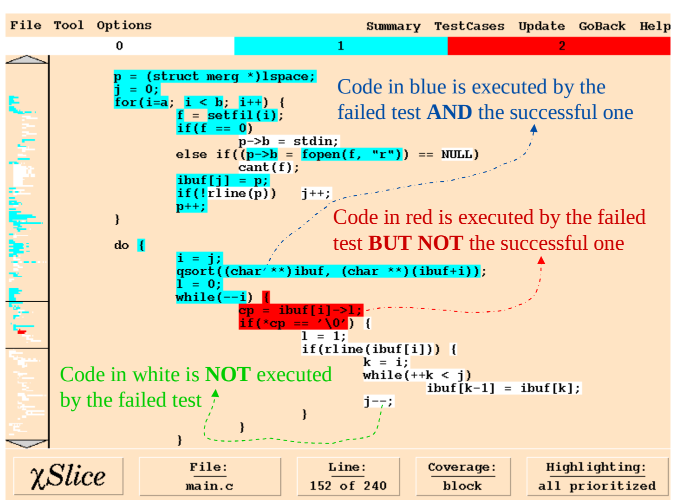
A test case in green runs the program successfully

A test case in red fails the program

Locating Bugs using Execution Dice (2)



Locating Bugs using Execution Dice (3)



Multiple Failed and Successful Tests (1)

- The more that successful tests execute a piece of code, the less likely for it to contain any fault.
- The more that failed tests with respect to a given fault execute a piece of code, the more likely for it to contain this fault.
- A piece of code containing a specific fault is
  - *inversely proportional* to the number of *successful tests* that execute it
  - *proportional* to the number of *failed tests* (with respect to this fault) that execute it.

W. E. Wong, T. Sugeta, Y. Qi, and J. C. Maldonado, "Smart Debugging Software Architectural Design in SDL," *Journal of Systems and Software,* Volume 76, Number 1, pp. 15-28, April 2005

Jump to Slide 97

Multiple Failed and Successful Tests (2)

- Need to consider *precision* and *recall*
  - intersection of failed tests – union of successful tests
  - union of failed tests – union of successful tests
  - intersection of failed tests – intersection of successful tests
  - union of failed tests – intersection of successful tests

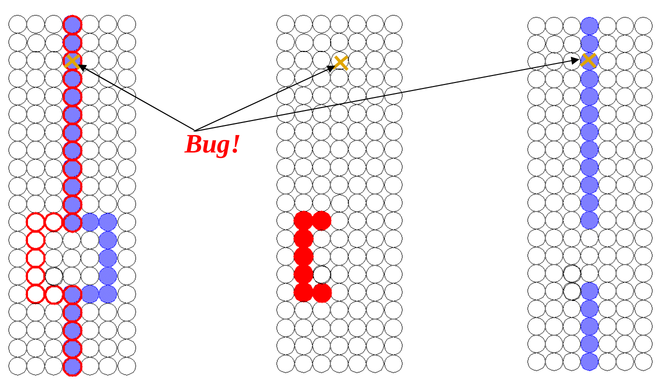
More Advanced Heuristics

- A bad dice does not contain the bug
  - *Augmentation of a bad* execution dice using *inter-block data dependency*
- A good dice with too much code
  - *Refining a good* execution dice using additional successful tests

W. E. Wong and Yu Qi, "An Execution Slice and Inter-Block Data Dependency-Based Approach for Fault Localization," *Journal of Systems and Software,* Volume 79, Number 7, pp. 891-903, July 2006

Augmentation of A Bad Execution Dice $\mathcal{D}^{(1)}$ (1)

- Bug is not in the execution dice
- Much code that is executed by both the failed test (the red path) and the successful test (the blue path)
- How to prioritize the code that still needs to be examined

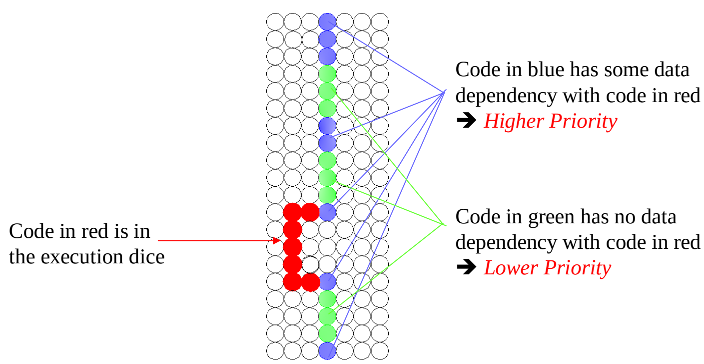


Augmentation of A Bad Execution Dice $\mathcal{D}^{(1)}$ (2)

- If the bug is not in $\mathcal{D}^{(1)}$, we need to examine additional code from the *rest* of the failed execution slice (i.e., $\mathcal{E}_F - \mathcal{D}^{(1)}$ denoted by Φ)
  - For a block β, the notation $\beta \in \Phi$ implies β is in the failed execution slice $\mathcal{E}_F$ but not in $\mathcal{D}^{(1)}$.
- More prioritization based on *inter-block data dependency*
- Define a "*direct data dependency*" relation Δ between a block β and an execution dice $\mathcal{D}^{(1)}$ such that $\beta \; \Delta \; \mathcal{D}^{(1)}$

> if and only if β defines a variable x that is used in $\mathcal{D}^{(1)}$ or β uses a variable y defined in $\mathcal{D}^{(1)}$.

Augmentation of A Bad Execution Dice $\mathcal{D}^{(1)}$ (3)



Augmentation of A Bad Execution Dice $\mathcal{D}^{(1)}$ (4)

- Construct $\mathcal{A}^{(1)}$, the augmented code segment from the first iteration, such that $\mathcal{A}^{(1)} = \{\beta \mid \beta \in \Phi \wedge (\beta \,\Delta\, \mathcal{D}^{(1)})\}$.
- set $k = 1$
- Examine code in $\mathcal{A}^{(k)}$ to see whether it contains the bug (🡐)
- If YES,

  then
  - STOP because we have located the bug

  else
  - set $k = k + 1$
- Construct $\mathcal{A}^{(k)}$, the augmented code segment from the k^{th} iteration, such that $\mathcal{A}^{(k)} = \mathcal{A}^{(k-1)} \cup \{\beta \mid b \in \Phi \wedge (\beta \,\Delta\, \mathcal{A}^{(k-1)})\}$.
- If $\mathcal{A}^{(k)} = \mathcal{A}^{(k-1)}$ (i.e., no new code can be included from the $(k\text{-}1)^{th}$ iteration to the k^{th} iteration)

  then
  - STOP

    At this point we have $\mathcal{A}^{(*)}$, the final augmented code segment, which equals $\mathcal{A}^{(k)}$ (and $\mathcal{A}^{(k-1)}$ as well)

  else
  - Go back to step (🡐)

Augmentation of A Bad Execution Dice $\mathcal{D}^{(1)}$ (5)

```
     #include <stdio.h>

     int main() {
     float a, b, c, d, x, y;

S0   scanf ("%f %f", &a, &b);
S1   if (a <= 0)
S2       c = 2*a + 1;
     else
S3       c = 3*a;
S4   if (b <= 0)
S5       d = b*b - 4*a*c
     else
S6       d = 5*b;
S7   x = b + d;
S8   y = c + d;
S9   printf ("x = %f  & y = %f \n", x, y);
     }
```

(a) the execution slice with respect to a failed test t_1 (a=3; b=5)

```
     #include <stdio.h>

     int main() {
     float a, b, c, d, x, y;

S0   scanf ("%f %f", &a, &b);
S1   if (a <= 0)
S2       c = 2*a + 1;
     else
S3       c = 3*a;
S4   if (b <= 0)
S5       d = b*b - 4*a*c
     else
S6       d = 5*b;
S7   x = b + d;
S8   y = c + d;
S9   printf ("x = %f  & y = %f \n", x, y);
     }
```

(b) the execution slice with respect to a successful test t_2 (a= -3; b=5)

Augmentation of A Bad Execution Dice $\mathcal{D}^{(1)}$ **(6)**

```
    #include <stdio.h>

    int main() {
    float a, b, c, d, x, y;

S0  scanf ("%f %f", &a, &b);
S1  if (a <= 0)
S2      c = 2*a + 1;
    else
S3      c = 3*a;
S4  if (b <= 0)
S5      d = b*b - 4*a*c
    else
S6      d = 5*b;
S7  x = b + d;
S8  y = c + d;
S9  printf ("x = %f  & y = %f \n", x, y);
    }
```

dice obtained by subtracting the execution slice in (b) from the execution slice in (a)

```
    #include <stdio.h>

    int main() {
    float a, b, c, d, x, y;

S0  scanf ("%f %f", &a, &b);
S1  if (a <= 0)
S2      c = 2*a + 1;
    else
S3      c = 3*a;
S4  if (b <= 0)
S5      d = b*b - 4*a*c
    else
S6      d = 5*b;
S7  x = b + d;
S8  y = c + d;
S9  printf ("x = %f  & y = %f \n", x, y);
    }
```

Bug! Should be 2*c

Code that has direct data dependency with S_3 (i.e., code in the dice)

Refining of A Good Execution Dice $\mathcal{D}^{(1)}$

- Construct the execution slices (denoted by $\Theta_1, \Theta_2, \ldots, \Theta_k$) with respect to successful tests $t_1, t_2, \ldots,$and t_k
- $\mathcal{D}^{(1)} = \mathcal{E}_F - \Theta_1$
- $\mathcal{D}^{(2)} = \mathcal{D}^{(1)} - \Theta_2 = \mathcal{E}_F - \Theta_1 - \Theta_2$
- We have $\mathcal{D}^{(1)} \supseteq \mathcal{D}^{(2)} \supseteq \mathcal{D}^{(3)}$, etc.
- Since we want to *examine the more suspicious code before the less suspicious code*, code in $\mathcal{D}^{(2)}$ should be examined before code in $\mathcal{D}^{(1)}$ but not in $\mathcal{D}^{(2)}$

```
for(i=0;i<Nmax;i++)
    { probtab[i].valore=0;
      probtab[i].posizione=0;}
i=0;
while (i<riga)
{ fscanf(fp,"%c",&c);
   for(j=0;c!='\n';j++)
   {fscanf(fp,"%c",&c);}
    i=i+1;}
fscanf(fp,"%d ",&a);
for (i=0; i<a; i++)
  { fscanf(fp,"%d ",&probtab[i].posizione);
   fscanf(fp,"%lf ",&probtab[i].valore);}
fclose(fp);
for(i=0;i<100;i++)
    urna[i]=0;
q=0; z=0; j=0;
for (i=0; i<Nmax; i++)
   { z=probtab[i].valore*100+z;
     for (j=q; j<z; j++)
     {urna[j]=probtab[i].posizione;}
     q=j;}
```

Part (a) Code in $\mathcal{D}^{(1)}$ is highlighted in red

```
for(i=0;i<Nmax;i++)
    { probtab[i].valore=0;
      probtab[i].posizione=0;}
i=0;
while (i<riga)
{ fscanf(fp,"%c",&c);
   for(j=0;c!='\n';j++)
   {fscanf(fp,"%c",&c);}
    i=i+1;}
fscanf(fp,"%d ",&a);
for (i=0; i<a; i++)
  { fscanf(fp,"%d ",&probtab[i].posizione);
   fscanf(fp,"%lf ",&probtab[i].valore);}
fclose(fp);
for(i=0;i<100;i++)
    urna[i]=0;
q=0; z=0; j=0;
for (i=0; i<Nmax; i++)
   { z=probtab[i].valore*100+z;
     for (j=q; j<z; j++)
     {urna[j]=probtab[i].posizione;}
     q=j;}
```

Part (b) Code in $\mathcal{D}^{(2)}$ is highlighted in red

```
for(i=0;i<Nmax;i++)
    { probtab[i].valore=0;
      probtab[i].posizione=0;}
i=0;
while (i<riga)
{ fscanf(fp,"%c",&c);
   for(j=0;c!='\n';j++)
   {fscanf(fp,"%c",&c);}
    i=i+1;}
fscanf(fp,"%d ",&a);
for (i=0; i<a; i++)
  { fscanf(fp,"%d ",&probtab[i].posizione);
   fscanf(fp,"%lf ",&probtab[i].valore);}
fclose(fp);
for(i=0;i<100;i++)
    urna[i]=0;
q=0; z=0; j=0;
for (i=0; i<Nmax; i++)
   { z=probtab[i].valore*100+z;
     for (j=q; j<z; j++)
     {urna[j]=probtab[i].posizione;}
     q=j;}
```

Part (c) Code in $\mathcal{D}^{(3)}$ is highlighted in red

An Incremental Approach

- Assume
  - debugging as soon as a failure is detected (i.e., only one failed test)
  - *n* (say 3) successful tests
- Assume the bug is in the code which is executed by the failed test but not the successful test(s)
  - first examining the code in $\mathcal{D}^{(3)}$ followed by code in $\mathcal{D}^{(2)}$ but not in $\mathcal{D}^{(3)}$, then code in $\mathcal{D}^{(1)}$ but not in $\mathcal{D}^{(2)}$
- If this assumption does not hold (i.e., the bug is not in $\mathcal{D}^{(1)}$), then we need to inspect additional code in the failed execution slice but not in $\mathcal{D}^{(1)}$
  - then starting with code in $\mathcal{A}^{(1)}$ but not in $\mathcal{D}^{(1)}$, followed by $\mathcal{A}^{(2)}$ but not in $\mathcal{A}^{(1)}$, …
- Prioritize code in a failed execution slice based on its likelihood of containing the bug. The prioritization is done by first using the refining method and then the augmentation method.
  - Examining code in $\mathcal{D}^{(3)}$, $\mathcal{D}^{(2)}$ but not in $\mathcal{D}^{(3)}$, $\mathcal{D}^{(1)}$ but not in $\mathcal{D}^{(2)}$, $\mathcal{A}^{(1)}$ but not in $\mathcal{D}^{(1)}$, $\mathcal{A}^{(2)}$ but not in $\mathcal{A}^{(1)}$, $\mathcal{A}^{(3)}$ but not in $\mathcal{A}^{(2)}$, … etc.
- In the worst case, we have to examine all the code in the failed execution slice.

Outline

- Motivation and Background
- Execution Dice-based Fault Localization
- Suspiciousness Ranking-based Fault Localization
  - Program Spectra-based Fault Localization
  - Code Coverage-based Fault Localization
  - Statistical Analysis-based Fault Localization
  - Neural Network-based Fault Localization
  - Similarity Coefficient-based Fault Localization
- Empirical Evaluation
- Theoretical Comparison: Equivalence
- Mutation-based Automatic Bug Fixing
- Conclusions

Suspiciousness Ranking-based Fault Localization

Overview

- Compute the suspiciousness (likelihood of containing bug) of each statement
- Rank all the executable statements in descending order of their suspiciousness
- Examine the statements one-by-one from the top of the ranking until the first faulty statement is located
- Statements with higher suspiciousness should be examined before statements with lower suspiciousness as the former are more likely to contain bugs than the latter

Techniques for Computing Suspiciousness

- Code coverage-based and calibration
- Crosstab: statistical analysis-based
- BP (Back Propagation) & RBF (Radial Basis Function) neural network
- Similarity coefficient-based
- Tarantula: heuristic-based
- SOBER: statistical analysis-based
- Liblit: statistical analysis-based

Take advantage of code coverage (namely, execution slice) and execution result of each test (success or failure) for debugging.

Outline

- Motivation and Background
- Execution Dice-based Fault Localization
- Suspiciousness Ranking-based Fault Localization
  - Program Spectra-based Fault Localization
  - Code Coverage-based Fault Localization
  - Statistical Analysis-based Fault Localization
  - Neural Network-based Fault Localization
  - Similarity Coefficient-based Fault Localization
- Empirical Evaluation
- Theoretical Comparison: Equivalence
- Mutation-based Automatic Bug Fixing
- Conclusions

Spectra-based Fault Localization

Spectra-Based Fault Localization Techniques

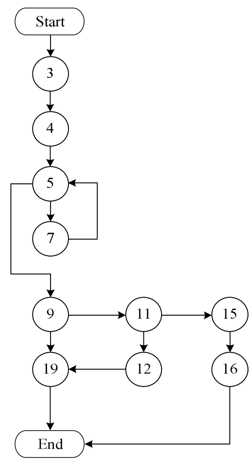
- Possible Program Spectra

	Name	Description
BHS	Branch Hit Spectra	conditional branches that are executed
BCS	Branch Count Spectra	number of times each conditional branch is executed
CPS	Complete Path Spectra	complete path that is executed
PHS	Path Hit Spectra	loop-free path that is executed
PCS	Path Count Spectra	number of times each loop-free path is executed
DHS	Data-Dependence Hit Spectra	definition-use pairs that are executed
DCS	Data-Dependence Count Spectra	number of times each definition-use pair is executed
OPS	Output Spectra	output that is produced
ETS	Execution Trace Spectra	execution trace that is produced
DVS	Data Value Spectra	the values of variables in the execution
ESHS	Executable Statement Hit Spectra	executable statements that are executed

A Sample Program for Program Spectra

- Given an integer n and a real number x, the program calculates x^n

```
1    double power (double x, int n)
2    {
3      int i;
4      int rv = 1;
5      for (i=0; i<abs(n); i++)
6      {
7        rv = rv × x;
8      }
9      if (n<0)
10     {
11       if (x!=0)
12         rv = 1/rv;
13       else
14       {
15         printf ("Error input.\n");
16         return 0;
17       }
18     }
19     return rv;
20   }
```



Return

Branch Hit Spectra

- BHS records the *conditional branches* that are covered by the test execution
- Suppose there are m conditional branches: b_1, b_2, …, b_m
- The spectrum with respect to b_i (i = 1, 2, …, m) indicates whether b_i is covered by the test execution
- There are 6 branches in the sample program:(5,7), (5,9), (9,19), (9,11), (11,12), and (11,15)
- When test case (x = 2, n = 3) is executed,

  the branch hit spectrum is (Y, Y, Y, N, N, N)

(11,15) is not covered

(11,12) is not covered

(5,7) is covered

(9,11) is not covered

(5,9) is covered

(9,19) is covered

Return

Branch Count Spectra

- BCS records the *number of times that each conditional branch* is executed
- Suppose there are m conditional branches: $b_1, b_2, \ldots, b_m$. The spectrum with respect to b_i ($i = 1, 2, \ldots, m$), denoted by s_i, indicates that b_i is executed s_i times by the test execution
- When test case (2,3) is executed, the *branch count spectrum* is (3, 1, 1, 0, 0, 0)

(5,7) is executed 3 times

(9,19) is executed one time

(11,15) is not executed

Return

Complete Path Spectra

- CPS records the *complete paths* that are traversed by the test execution
- When test case (2,3) is executed, the CPS is $(3,4,(5,7)^3,9,19)$

Statement 5 and 7 are executed 3 times

Return

Path Hit Spectra

- PHS records the *intra-procedural, loop-free paths* that are covered by the test execution
- The sample program has six possible paths
  - 3,4,5,9,19
  - 3,4,5,7,9,19
  - 3,4,5,9,11,12,19
  - 3,4,5,7, 9,11,12,19
  - 3,4,5,9,11,15,16
  - 3,4,5,7,9,11,15,16
- With respect to the execution of test case (2,3), the *path hit spectrum* can be represented by
  - (Y,N,N,N,N,N)

(3,4,5,9,19) is covered

(3,4,5,7,9,11,15,16) is not covered

[Return]

Path Count Spectra

- PCS records the *number of times that each intra-procedural, loop-free path* is covered by the test execution
- The sample program has six possible loop-free paths
  - 3,4,5,9,19
  - 3,4,5,7,9,19
  - 3,4,5,9,11,12,19
  - 3,4,5,7, 9,11,12,19
  - 3,4,5,9,11,15,16
  - 3,4,5,7,9,11,15,16
- When test case (2,3) is executed, the *path count spectrum* can be represented by
  - (1,0,0,0,0,0)
  - When the function is executed more than one time, the elements in PCS may be larger than 1

(3,4,5,9,19) is executed one time

(3,4,5,7,9,11,15,16) is not executed

[Return]

Data-Dependence Hit Spectra

- DHS records the *definition-use pairs* that are covered by the execution
- With respect to the sample program, let's focus on the following definition-use pairs
  - (rv, 4, 7)
  - (rv, 4, 19)
  - (rv, 7, 7)
  - (rv, 7, 12)
  - (rv, 7, 19)
  - (rv, 12, 19)
- When test case (2,3) is executed, the spectrum can be represented by
  - (Y,N,Y,N,Y,N) which implies (rv,4,7), (rv,7, 7) and (rv,7,19) are covered by this execution

Return

Data-Dependence Count Spectra

- DCS records the *number of times that each definition-use pair* is executed
- With respect to the sample program, let's focus on the following definition-use pairs
  - (rv, 4, 7)
  - (rv, 4, 19)
  - (rv, 7, 7)
  - (rv, 7, 12)
  - (rv, 7, 19)
  - (rv, 12, 19)
- When test case (2,3) is executed, the *data-dependence count spectrum* can be represented by (1,0,2,0,1,0)

(rv, 7, 7) is executed 2 times

Return

Output Spectra

- OPS records the *outputs produced* by the test executions
- With respect to the sample program, when test case (2,3) is executed, the *output spectrum* can be represented by a value 8, which is the output of the function

Return

Execution Trace Spectra

- ETS records the *sequence of each program statement* traversed by the test execution
- With respect to the sample program, when case (2,3) is executed, the execution trace spectrum can be represented by

  (int i, double rv = 1, (for(i=0;i<abs(n);i++), rv = rv * x)^3, if(n<0),return rv)
- Difference between ETS and CPS (Complete Path Spectrum):
  - ETS records the actual instructions, whereas CPS does not

These statements are executed 3 times

Return

Data Value Spectra

- DVS records the *values of variables*
- With respect to the sample program, we focus on the value of variable *rv*
  - When test case (2,3) is executed, the sequence of the values of *rv* is (1,2,4,8) which is one of the DVS representations

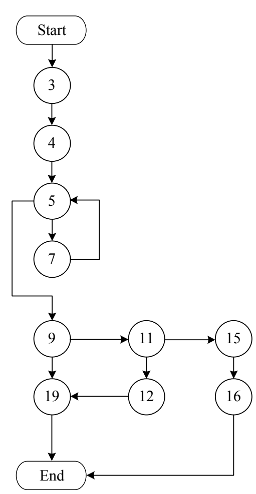
Return

Executable Statement Hit Spectra

- ESHS records the *executable statements that are covered* by the test execution.
  - excluding comments, blank lines, (some) variable declarations, function declarations, etc.
- Suppose there are m executable statements: s_1, s_2, …, s_m
- The spectrum with respect to s_i (i = 1, 2, …, m), indicates whether s_i is covered by the test execution.
- There are 9 executable statements at lines 4, 5, 7, 9, 11, 12, 15,16 and 19
- When test case (2,3) is executed, the *executable statement hit spectrum* is
  (Y, Y, Y, Y, N, N, N, N, Y).

Statement 4 is executed

Statement 11 is not executed



Next Return

Outline

- Motivation and Background
- Execution Dice-based Fault Localization
- Suspiciousness Ranking-based Fault Localization
  - Program Spectra-based Fault Localization
  - Code Coverage-based Fault Localization
  - Statistical Analysis-based Fault Localization
  - Neural Network-based Fault Localization
  - Similarity Coefficient-based Fault Localization
- Empirical Evaluation
- Theoretical Comparison: Equivalence
- Mutation-based Automatic Bug Fixing
- Conclusions

Code Coverage-based Fault Localization & Calibration

W. E. Wong, V. Debroy and B. Choi, "A Family of Code Coverage-based Heuristics for Effective Fault Localization," *Journal of Systems and Software*, Volume 83, Issue 2, pp. 188-208, February 2010
(Best Paper Award; COMPSAC 2007)

Code Coverage-based & Calibration (1)

- Suppose for a large test suite, say 1000 test cases, a majority of them, say 995, are successful test cases and only a small number of failed test cases (five in this example) will cause an execution failure.
- The challenge is how to use these five failed tests and the 995 successful tests to conduct an effective debugging.

- How can each additional test case that executes the program successfully help locate program bugs?
- What about each additional test case that makes the program execution fail?

Code Coverage-based & Calibration (2)

- Should all the successful test executions provide the same contribution to locate software bugs?
- Intuitively, the answer should be "no"
- If a piece of code has already been executed successfully 994 times, then the contribution of the 995th successful execution is likely to be less than, for example, the contribution of the second successful execution when the code is only executed successfully once

- We propose that with respect to a piece of code, the contribution introduced by the first successful test that executes it in computing its likelihood of containing a bug is larger than or equal to that of the second successful test that executes it, which is larger than or equal to that of the third successful test that executes it, etc.

- The same also applies to the failed tests.

Code Coverage-based & Calibration (3)

Φ_F	total number of failed test cases for $\mathcal{B}$
Φ_S	total number of successful test cases for $\mathcal{B}$
$\mathcal{N}_F$	total number of failed test cases with respect to $\mathcal{B}$ that execute $\mathcal{S}$
$\mathcal{N}_S$	total number of successful test cases that execute $\mathcal{S}$
$c_{F,i}$	contribution from the i^{th} failed test case that executes $\mathcal{S}$
$c_{S,i}$	contribution from the i^{th} successful test case that executes $\mathcal{S}$
$\mathcal{G}_F$	number of groups for the failed tests that execute $\mathcal{S}$
$\mathcal{G}_S$	number of groups for the successful tests that execute $\mathcal{S}$
$n_{F,i}$	maximal number of failed test cases in the i^{th} failed group
$n_{S,i}$	maximal number of successful test cases in the i^{th} successful group
$w_{F,i}$	contribution from each test in the i^{th} failed group
$w_{S,i}$	contribution from each test in the i^{th} successful group
$\chi_{F/S}$	Φ_F/Φ_S

- $c_{S,1} \geq c_{S,2} \geq c_{S,3} \geq \ldots \geq c_{S,\mathcal{N}_S}$ and $c_{F,1} \geq c_{F,2} \geq c_{F,3} \geq \ldots \geq c_{F,\mathcal{N}_F}$
- If the statement $\mathcal{S}$ is executed by at least one failed test, then the total contribution from all the successful tests that execute $\mathcal{S}$ should be less than the total contribution from all the failed tests that execute $\mathcal{S}$ (namely, $\sum_{i=1}^{\mathcal{N}_S} c_{S,i} < \sum_{k=1}^{\mathcal{N}_F} c_{F,k}$)
- All the tests in the same failed group have the same contribution towards fault localization, but tests from different groups have different contributions

Code Coverage-based & Calibration (4)

- For illustrative purposes, we set $\mathcal{G}_F = \mathcal{G}_S = 3$, $n_{F,1} = n_{S,1} = 2$, and $n_{F,2} = n_{S,2} = 4$
  - The first failed (or successful) group has at most two tests, the second group has at most four from the remaining, and the third has everything else, if any.
- We also assume each test case in the first, second, and third failed groups gives a contribution of 1, 0.1 and 0.01, respectively ($w_{F,1} = 1$, $w_{F,2} = 0.1$, and $w_{F,3} = 0.01$).
- Similarly, we set $w_{S,1} = 1$, $w_{S,2} = 0.1$, and $w_{S,3}$ to be a small value defined as $\alpha \times \chi_{F/S}$ where α is a scaling factor.

$$\left[(1.0)\times n_{F,1} + (0.1)\times n_{F,2} + (0.01)\times n_{F,3}\right] - \left[(1.0)\times n_{S,1} + (0.1)\times n_{S,2} + \alpha\times\chi_{F/S}\times n_{S,3}\right]$$

where $n_{F,1} = \begin{cases} 0, & \text{for } \mathcal{N}_F = 0 \\ 1, & \text{for } \mathcal{N}_F = 1 \\ 2, & \text{for } \mathcal{N}_F \geq 2 \end{cases}$ $n_{F,2} = \begin{cases} 0, & \text{for } \mathcal{N}_F \leq 2 \\ \mathcal{N}_F - 2, & \text{for } 3 \leq \mathcal{N}_F \leq 6 \\ 4, & \text{for } \mathcal{N}_F > 6 \end{cases}$ $n_{F,3} = \begin{cases} 0, & \text{for } \mathcal{N}_F \leq 6 \\ \mathcal{N}_F - 6, & \text{for } \mathcal{N}_F > 6 \end{cases}$ and

$n_{S,1} = \begin{cases} 0, & \text{for } n_{F,1} = 0, 1 \\ 1, & \text{for } n_{F,1} = 2 \text{ and } \mathcal{N}_S \geq 1 \end{cases}$ $n_{S,2} = \begin{cases} 0, & \text{for } \mathcal{N}_S \leq n_{S,1} \\ \mathcal{N}_S - n_{S,1}, & \text{for } n_{S,1} < \mathcal{N}_S < n_{F,2} + n_{S,1} \\ n_{F,2}, & \text{for } \mathcal{N}_S \geq n_{F,2} + n_{S,1} \end{cases}$ $n_{S,3} = \begin{cases} 0, & \text{for } \mathcal{N}_S < n_{S,1} + n_{S,2} \\ \mathcal{N}_S - n_{S,1} - n_{S,2}, & \text{for } \mathcal{N}_S \geq n_{S,1} + n_{S,2} \end{cases}$

Code Coverage-based & Calibration (5)

	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	s_9	s_{10}	r	
t_1	1	0	0	0	1	1	0	1	1	1	1	failed test
t_2	1	1	0	1	0	0	0	0	1	0	0	
t_3	1	0	1	0	0	1	1	1	1	1	1	failed test
t_4	1	1	1	1	0	0	1	1	1	0	0	
t_5	1	1	1	1	0	1	1	1	1	0	0	
t_6	1	0	0	0	0	0	1	1	1	1	0	
t_7	1	1	0	0	0	1	1	0	1	1	0	
t_8	1	1	1	1	0	0	1	1	1	0	0	
t_9	0	1	0	0	0	1	0	0	1	1	0	
t_{10}	1	0	0	0	1	0	1	1	1	1	1	failed test
t_{11}	0	1	1	0	0	0	0	1	1	0	0	
t_{12}	1	1	0	1	1	0	1	0	1	1	0	
t_{13}	1	0	1	0	0	0	1	1	1	0	0	
t_{14}	0	1	0	1	1	0	1	0	1	1	0	
t_{15}	1	0	1	0	1	0	1	1	1	1	1	failed test
t_{16}	0	1	0	1	0	0	0	1	1	0	0	
t_{17}	1	1	1	1	0	0	1	1	1	0	0	
t_{18}	1	0	1	0	1	1	0	1	1	1	1	failed test
t_{19}	1	1	1	1	0	0	1	0	1	0	0	
t_{20}	0	0	1	1	1	1	1	0	1	1	0	
number of failed tests that execute each statement	5	0	3	0	4	3	3	5	5	5		
number of successful tests that execute each statement	10	12	8	10	3	4	11	8	15	6		
suspiciousness of each statement	0.980	-0.04	0.980	-0.033	1.000	0.993	0.970	0.987	0.963	0.993		

More Details

Next

Code Coverage-based & Calibration (6)

- Two fundamental principles
  - $c_{S,1} \geq c_{S,2} \geq c_{S,3} \geq \ldots \geq c_{S,\mathcal{N}_S}$ and $c_{F,1} \geq c_{F,2} \geq c_{F,3} \geq \ldots \geq c_{F,\mathcal{N}_F}$
  - $\sum_{i=1}^{\mathcal{N}_S} c_{S,i} < \sum_{k=1}^{\mathcal{N}_F} c_{F,k}$

Outline

- Motivation and Background
- Execution Dice-based Fault Localization
- Suspiciousness Ranking-based Fault Localization
  - Program Spectra-based Fault Localization
  - Code Coverage-based Fault Localization
  - Statistical Analysis-based Fault Localization
  - Neural Network-based Fault Localization
  - Similarity Coefficient-based Fault Localization
- Empirical Evaluation
- Theoretical Comparison: Equivalence
- Mutation-based Automatic Bug Fixing
- Conclusions

Crosstab-based Fault Localization

W. Eric Wong, Vidroha Debroy and Dianxiang Xu, “Towards Better Fault Localization: A Crosstab-based Statistical Approach,” *IEEE Transactions on Systems, Man, and Cybernetics – Part C: Applications & Reviews*
(Accepted in December 2010 for publication)
(http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=05772029)

Crosstab

- The crosstab (*cross-classification table*) analysis is used to study

the relationship between two or more categorical variables.

- A crosstab is constructed for each statement as follows

	ω is covered	ω is not covered	Σ
successful executions	$N_{CS}(\omega)$	$N_{US}(\omega)$	N_S
failed executions	$N_{CF}(\omega)$	$N_{UF}(\omega)$	N_F
Σ	$N_C(\omega)$	$N_U(\omega)$	N

N	total number of test cases
N_F	total number of failed test cases
N_S	total number of successful test cases
$N_C(\omega)$	number of test cases covering ω
$N_{CF}(\omega)$	number of failed test cases covering ω
$N_{CS}(\omega)$	number of successful test cases covering ω
$N_U(\omega)$	number of test cases not covering ω
$N_{UF}(\omega)$	number of failed test cases not covering ω
$N_{US}(\omega)$	number of successful test cases not covering ω

Dependency Relationship (1)

- For each crosstab, we conduct a *hypothesis test* to check the *dependency relationship*. The null hypothesis is

H_0: Program execution result is independent of the coverage of statement ω

- A *chi-square test* can be used to determine whether this hypothesis should be rejected. The *Chi-square statistic* is given by

$$\chi^2(\omega) = \frac{(N_{CF}(\omega) - E_{CF}(\omega))^2}{E_{CF}(\omega)} + \frac{(N_{CS}(\omega) - E_{CS}(\omega))^2}{E_{CS}(\omega)} + \frac{(N_{UF}(\omega) - E_{UF}(\omega))^2}{E_{UF}(\omega)} + \frac{(N_{US}(\omega) - E_{US}(\omega))^2}{E_{US}(\omega)} \quad (1)$$

where $E_{CF}(\omega) = \frac{N_C(\omega) \times N_F}{N}$, $E_{CS}(\omega) = \frac{N_C(\omega) \times N_S}{N}$, $E_{US}(\omega) = \frac{N_U(\omega) \times N_S}{N}$, and $E_{UF}(\omega) = \frac{N_U(\omega) \times N_F}{N}$,

- Under the null hypothesis, the statistic $\chi^2(\omega)$ has approximately a *Chi-square distribution*.

Dependency Relationship (2)

- Given a level of significance σ (for example, 0.05), we can find the corresponding *Chi-square critical value* χ^2_σ, from the Chi-square distribution table.
  - If $\chi^2(\omega) > \chi^2_\sigma$, we reject the null hypothesis, i.e., the execution result is dependent on the coverage of ω.
  - Otherwise, we accept the null hypothesis, i.e., the execution result and the coverage of ω are "independent."

Degree of Association (1)

- The "dependency" relationship indicates a high association among the variables, whereas the "independency" relationship implies a low association.
- Instead of the so-called "dependency"/ "independency" relationship, we are more interested in the degree of association between the execution result and the coverage of each statement.
- This degree can be measured based on the standard Chi-square statistic. However, such a measure increases with increasing sample size. As a result, the measure by itself may not give the "true" degree of association.
- One way to fix this problem is to use the *contingency coefficient* computed as follows

$$\mathcal{M}(\omega) = \frac{\chi^2(\omega)/N}{\sqrt{(row-1)(col-1)}} \qquad (2)$$

where *row* and *col* are the number of categorical variables in all rows and columns, respectively, of the crosstab

Degree of Association (2)

- The contingency coefficient $\mathcal{M}(\omega)$ lies between 0 and 1.
  - When $\chi^2(\omega) = 0$, it has the lower limit 0 for complete independence.
  - In the case of complete association, the coefficient can reach the upper limit 1 when *row* = *col*
- In our case, *row* = *col* = 2 and *N* is fixed. From Equation (2), $\mathcal{M}(\omega)$ increases with increasing $\chi^2(\omega)$. ⓘ
- Under this condition, the Chi-square statistic $\chi^2(\omega)$ for statement ω gives a good indication of the degree of the association between the execution result and the coverage of ω.
  - *N* is fixed because every faulty version is executed with respect to all the test cases

What kind of Execution Result is More Associated (1)

- Need to decide whether it is the failed or the successful execution result that is more associated with the coverage of the statement.
- For each statement ω, we compute $\mathcal{P}_F(\omega)$ and $\mathcal{P}_S(\omega)$ as $\frac{N_{CF}(\omega)}{N_F}$ and $\frac{N_{CS}(\omega)}{N_S}$ which are the percentages of all failed and successful tests that execute ω.
- If $\mathcal{P}_F(\omega)$ is larger than $\mathcal{P}_S(\omega)$, then the association between the failed execution and the coverage of ω is higher than that between the successful execution and the coverage of ω.

What kind of Execution Result is More Associated (2)

- We define $\varphi(\omega)$ as

$$\varphi(\omega) = \frac{\mathcal{P}_F(\omega)}{\mathcal{P}_S(\omega)} = \frac{N_{CF}(\omega)/N_F}{N_{CS}(\omega)/N_S} \qquad (3)$$

- If $\varphi(\omega) = 1$, we have $\chi^2(\omega) = 0$, which implies the execution result is completely independent of the coverage of ω. In this case, we say the coverage of ω makes the same contribution to both the failed and the successful execution result.

- If $\varphi(\omega) > 1$, the coverage of ω is more associated with the failed execution.
- If $\varphi(\omega) < 1$, the coverage of ω is more associated with the successful execution.

Five Classes of Statements

- Depending on the values of $\chi^2(\omega)$ and $\varphi(\omega)$, statements of the program being debugged can be classified into five classes:
  - Statements with $\varphi > 1$ and $\chi^2 > \chi^2_\sigma$, have a high degree of association between their coverage and the failed execution result
  - Statements with $\varphi > 1$ and $\chi^2 \leqslant \chi^2_\sigma$, have a low degree of association between their coverage and the failed execution result
  - Statements with $\varphi < 1$ and $\chi^2 > \chi^2_\sigma$, have a high degree of association between their coverage and the successful execution result
  - Statements with $\varphi < 1$ and $\chi^2 \leqslant \chi^2_\sigma$, have a low degree of association between their coverage and the successful execution result
  - Statements with $\varphi = 1$ (under this situation $0 = \chi^2 < \chi^2_\sigma$,) whose coverage is independent of the execution result

Statements in the first class are most likely (i.e., have the highest suspiciousness) to contain program bugs followed by those in the second, the fifth, and the fourth classes, respectively. Statements in the third class are least likely (i.e., have the least suspiciousness) to contain bugs.

Suspiciousness of Each Statement

- The larger the coefficient $\mathcal{M}(\omega)$, the higher the association between the execution result and the coverage of ω.
  - For statements in the first and the second classes, those with a larger $\mathcal{M}$ are more suspicious.
  - For statements in the third and the fourth classes, those with a smaller $\mathcal{M}$ are more suspicious.
- The suspiciousness of a statement ω can be defined by a statistic ζ as

$$\zeta(\omega) = \begin{cases} \mathcal{M}(\omega) & \text{if } \varphi(\omega) > 1 \\ 0 & \text{if } \varphi(\omega) = 1 \\ -\mathcal{M}(\omega) & \text{if } \varphi(\omega) < 1 \end{cases} \qquad (4)$$

- Each ζ lies between -1 and 1. The larger the ζ value, the more suspicious the statement ω.

Crosstab Example (1)

- The following table gives the statement coverage and execution results. Of the 36 test cases, there are nine failed tests (e.g., t_1) and 27 successful tests (e.g., t_2)
  - An entry 1 implies the statement is covered by the corresponding test and an entry 0 means it is not.
  - An entry 1 implies a failed execution and an entry 0 means a successful execution.

	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	s_9	s_{10}	r
t_1	1	0	0	0	1	1	0	1	1	1	1
t_2	1	1	0	1	0	0	0	0	1	0	0
t_3	1	0	1	0	0	1	1	1	1	1	1
t_4	1	1	1	1	0	0	1	1	1	0	0
t_5	1	1	1	1	0	1	1	1	1	0	0
t_6	1	0	0	0	0	0	1	1	1	1	0
t_7	1	1	0	0	0	1	1	0	1	1	0
t_8	1	1	1	1	0	0	1	1	1	0	0
t_9	0	1	0	0	0	1	0	0	1	1	0
t_{10}	1	0	0	0	1	0	1	1	1	1	1
t_{11}	0	1	1	0	0	0	0	1	1	0	0
t_{12}	1	1	0	1	1	0	1	0	1	1	0
t_{13}	1	0	1	0	0	0	1	1	1	0	0
t_{14}	0	1	0	1	1	0	1	0	1	1	0
t_{15}	1	0	1	0	1	0	1	1	1	1	1
t_{16}	0	1	0	1	0	0	0	1	1	0	0
t_{17}	1	1	1	1	0	0	1	1	1	0	0
t_{18}	1	0	1	0	1	1	0	1	1	1	1
t_{19}	1	1	1	1	0	0	1	0	1	0	0
t_{20}	0	0	1	1	1	1	1	0	1	1	0
t_{21}	1	1	0	0	0	0	1	1	0	1	1
t_{22}	0	0	0	1	1	0	1	1	0	0	0
t_{23}	0	1	1	0	0	0	1	0	0	1	0
t_{24}	1	1	1	0	1	0	0	0	0	1	0
t_{25}	1	0	0	1	1	1	0	1	1	0	1
t_{26}	1	1	1	0	0	1	0	0	0	1	0
t_{27}	1	0	0	0	0	1	1	0	1	1	0
t_{28}	0	1	1	1	1	0	0	1	0	0	0
t_{29}	0	0	1	1	1	1	0	1	0	0	0
t_{30}	1	0	0	0	1	1	0	0	1	1	0
t_{31}	1	1	0	1	0	0	0	1	0	0	1
t_{32}	1	0	0	1	1	1	1	0	0	1	0
t_{33}	1	1	0	0	0	1	0	1	0	0	0
t_{34}	1	0	1	1	1	0	0	1	1	0	1
t_{35}	0	0	0	1	1	0	1	1	1	0	0
t_{36}	0	0	1	0	1	0	0	0	1	0	0

Crosstab Example (2)

- We can construct the crosstab for s_1 as shown in the following

	s_1 is covered	s_1 is not covered	$\sum$
successful executions	16	11	27
failed executions	9	0	9
$\sum$	25	11	36

- We have $E_{CF}(s_1) = \frac{N_C(s_1) \times N_F}{N} = \frac{25 \times 9}{36} = 6.25,$

$$E_{CS}(s_1) = \frac{N_C(s_1) \times N_S}{N} = \frac{25 \times 27}{36} = 18.75,$$

$$E_{UF}(s_1) = \frac{N_U(s_1) \times N_F}{N} = \frac{11 \times 9}{36} = 2.75,$$

$$E_{US}(s_1) = \frac{N_U(s_1) \times N_S}{N} = \frac{11 \times 27}{36} = 8.25.$$

- From Equation (1)

$$\chi^2(s_1) = \frac{(N_{CF}(s_1) - E_{CF}(s_1))^2}{E_{CF}(s_1)} + \frac{(N_{CS}(s_1) - E_{CS}(s_1))^2}{E_{CS}(s_1)} + \frac{(N_{UF}(s_1) - E_{UF}(s_1))^2}{E_{UF}(s_1)} + \frac{(N_{US}(s_1) - E_{US}(s_1))^2}{E_{US}(s_1)}$$

$$= \frac{(9 - 6.25)^2}{6.25} + \frac{(16 - 18.75)^2}{18.75} + \frac{(0 - 2.75)^2}{2.75} + \frac{(11 - 8.25)^2}{8.25} = 5.2800$$

Crosstab Example (3)

- If we choose the level of significance as 0.05, the Chi-square critical value is 3.841. Since $\chi^2(s_1) = 5.2800$ is larger than 3.841, the null hypothesis for s_1 should be rejected.
- Similarly, we can compute χ^2 for other statements. For example, we have $\chi^2(s_2) = 4.4954$, $\chi^2(s_3) = 0.1481$, and $\chi^2(s_4) = 1.3333$.
- Next, we use Equation (2) to compute the *contingency coefficient* $\mathcal{M}$ for each statement. We have $\mathcal{M}(s_1)= 0.1467$, $\mathcal{M}(s_2)= 0.1249$, $\mathcal{M}(s_3)= 0.0041$, and $\mathcal{M}(s_4)= 0.0370$.
- Compute φ and ζ using Equations (3) and (4)

	χ^2	$\mathcal{M}$	φ	ζ
s_1	5.2800	0.1467	1.6875	0.1467
s_2	4.4954	0.1249	0.3529	−0.1249
s_3	0.1481	0.0041	0.8571	−0.0041
s_4	1.3333	0.0370	0.6000	−0.0370
s_5	1.8204	0.0506	1.6364	0.0506
s_6	0.1558	0.0043	1.2000	0.0043
s_7	0.6000	0.0167	0.7500	−0.0167
s_8	7.6364	0.2121	2.0769	0.2121
s_9	0.1846	0.0051	1.1053	0.0051
s_{10}	1.3333	0.0370	1.5000	0.0370

- Based on the suspiciousness, statement s_8 should be examined first for locating program bugs followed by s_1, s_5, s_{10}, s_9, s_6, s_3, s_7, s_4, and s_2.

Jump to Slide 164 Level of Significance

Outline

- Motivation and Background
- Execution Dice-based Fault Localization
- Suspiciousness Ranking-based Fault Localization
  - Program Spectra-based Fault Localization
  - Code Coverage-based Fault Localization
  - Statistical Analysis-based Fault Localization
  - Neural Network-based Fault Localization
  - Similarity Coefficient-based Fault Localization
- Empirical Evaluation
- Theoretical Comparison: Equivalence
- Mutation-based Automatic Bug Fixing
- Conclusions

RBF Neural Network-based Fault Localization

- W. Eric Wong, Vidroha Debroy, Richard Golden, Xiaofeng Xu and Bhavani Thuraisingham, "Effective Software Fault Localization using an RBF Neural Network," *IEEE Transactions on Reliability* (Accepted in May 2011 for publication) (http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6058639)
- W. Eric Wong and Yu Qi, "BP Neural Network-based Effective Fault Localization," *International Journal of Software Engineering and Knowledge Engineering*, 19(4): 573-597, June 2009

RBF Neural Network (1)

- A typical RBF neural network has a three-layer feed-forward structure
  - Input layer: Serve as an input distributor to the hidden layer by passing inputs to the hidden layer without changing their values.
  - Hidden layer: All neurons in this layer simultaneously receive the n-dimensional real-valued input vector x.
    - Each neuron uses a Radial Basis Function (RBF) as the activation function
    - An RBF is a strictly positive radically symmetric function, where the center μ has the unique maximum and the value drops off rapidly to zero away from the center
    - When the distance between x and μ (denoted as ||x–μ||) is smaller than the receptive field width σ, the function has an appreciable value.
    - A commonly used RBF is the Gaussian basis function

$$R_j(\mathbf{x}) = \exp\left(-\frac{\|\mathbf{x}-\boldsymbol{\mu}_j\|^2}{2\sigma_j^2}\right)$$

where μ_j and σ_j are the *mean* (namely, the *center*) and the *standard deviation* (namely, the *width*) of the receptive field of the j^{th} hidden layer neuron, and $R_j(\text{x})$ is the corresponding activation function.

RBF Neural Network (2)

- Usually the distance $\|\mathbf{x}-\boldsymbol{\mu}\|$ is the Euclidean distance between **x** and **μ** ($\|\mathbf{x}-\boldsymbol{\mu}\|_E$)
- However, ($\|\mathbf{x}-\boldsymbol{\mu}\|_E$) is inappropriate in the fault localization context
- We use a weighted bit-comparison based distance ($\|\mathbf{x}-\boldsymbol{\mu}\|_{WBC}$)

Let **x** be $\mathbf{c}_{t_i}$ (the coverage vector of i^{th} test case t_i)

$$\|\mathbf{c}_{t_i} - \boldsymbol{\mu}_j\|_{\text{WBC}} = \sqrt{1-\cos\theta_{\mathbf{c}_{t_i},\boldsymbol{\mu}_j}}$$

where $\cos\theta_{\mathbf{c}_{t_i},\boldsymbol{\mu}_j} = \dfrac{\mathbf{c}_{t_i} \bullet \boldsymbol{\mu}_j}{\|\mathbf{c}_{t_i}\|_E \|\boldsymbol{\mu}_j\|_E} = \dfrac{\sum_{k=1}^{m} (\mathbf{c}_{t_i})_k (\boldsymbol{\mu}_j)_k}{\sqrt{\sum_{k=1}^{m} [(\mathbf{c}_{t_i})_k]^2} \times \sqrt{\sum_{k=1}^{m} [(\boldsymbol{\mu}_j)_k]^2}}$,

where $(\mathbf{c}_{t_i})_k$ and $(\boldsymbol{\mu}_j)_k$ are the k^{th} element of $\mathbf{c}_{t_i}$ and $\boldsymbol{\mu}_j$, respectively.

This distance is more desirable because it effectively takes into account the number of bits that are *both* 1 in two coverage vectors (i.e., those statements covered by both vectors).

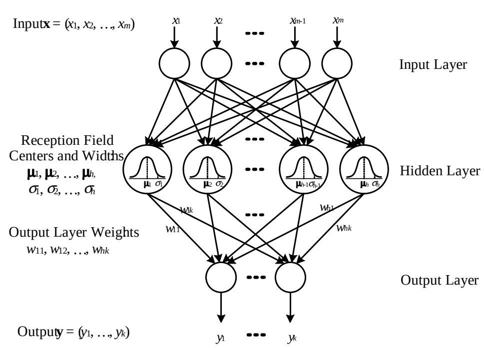
RBF Neural Network (3)

– Output layer: $\mathbf{y} = [y_1, y_2, \ldots, y_k]$ with y_i as the output of the i^{th} neuron given by

$$y_i = \sum_{j=1}^{h} w_{ji} R_j(\mathbf{x}) \qquad \text{for } i = 1, 2, \ldots, k$$

where h is the number of neurons in the hidden layer and w_{ji} is the *weight* associated with the link connecting the j^{th} hidden layer neuron and the i^{th} output layer neuron.

RBF Neural Network (4)



- An RBF network implements a mapping from the m dimensional real-valued input space to the k dimensional real-valued output space. In between, there is a layer of hidden-layer space.
- The transformation from the input space to the hidden-layer space is *nonlinear*, whereas the transformation from the hidden-layer space to the output space is *linear*.
- The parameters that need to be trained are the *centers* (i.e., $\mathbf{m}_1, \mathbf{m}_2, \ldots, \mathbf{m}_h$) and *widths* (i.e., $s_1, s_2, \ldots, s_h$) of the receptive fields of hidden layer neurons, and the *output layer weights*.

RBF Neural Network (5)

- We construct an RBF neural network with
  - m input layer neurons (each of which corresponds to one element in a given coverage vector of a test case)
  - one output layer neuron (corresponding to the execution result of test t_i)
  - one hidden layer between the input and output layers

RBF Neural Network (6)

- Once an RBF network is trained, it provides a good mapping between the input (the coverage vector of a test case) and the output (the corresponding execution result).

- It can then be used to identify suspicious code of a given program in terms of its likelihood of containing bugs.

- To do so, we use a set of *virtual* test cases $v_1, v_2, \ldots, v_m$ whose coverage vectors are where

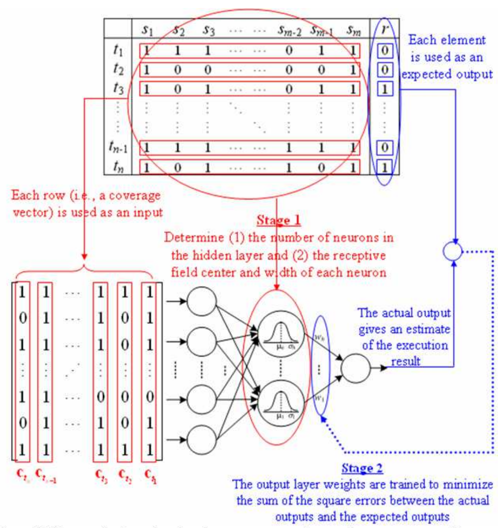
$$\begin{bmatrix} \mathbf{c}_{v_1} \\ \mathbf{c}_{v_2} \\ \vdots \\ \mathbf{c}_{v_m} \end{bmatrix} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix}$$

Note that execution of test v_j covers only one statement s_j

RBF Neural Network (7)

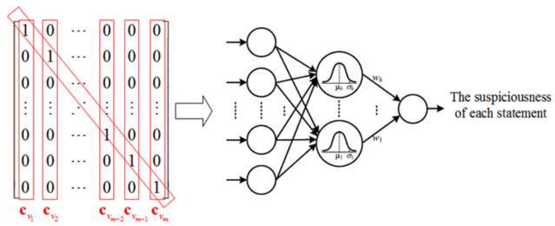
- If the execution of v_j fails, the probability that the bugs are contained in s_j is high.
- This suggests that during the fault localization, we should first examine the statements whose corresponding virtual test case fails.
- However, the execution results of these virtual tests can rarely be collected in the real world because it is very difficult, if not impossible, to construct such tests.
- When the coverage vector $\mathbf{c}_{v_j}$ of a virtual test case v_j is input to the trained neural network, its output $\hat{r}_{v_j}$ is the conditional expectation of whether the execution of v_j fails given $\mathbf{c}_{v_j}$.
- This implies the larger the value of $\hat{r}_{v_j}$ the more likely that the execution of v_j fails.
- Together, we have the larger the value of $\hat{r}_{v_j}$ the more likely it is that s_j contains the bug.
- We can treat $\hat{r}_{v_j}$ as the suspiciousness of s_j in terms of its likelihood of containing the bug.

Summary of RBF-based Fault Localization (1)



Train an RBF neural network using the coverage vectors and program execution results

Summary of RBF-based Fault Localization (2)



Compute the suspiciousness of each statement in P using virtual test cases

Three Novel Aspects

- Introduce a method for representing test cases, statement coverage, execution results within a modified RBF neural network formalism
  - Training with example test cases and execution results
  - Testing with virtual test cases
- Develop a novel algorithm to simultaneously estimate the number of hidden neurons and their receptive field centers
- Instead of using the *traditional Euclidean distance* which has been proved to be inappropriate in the fault localization context, a *weighted bit-comparison based distance* is defined to measure the distance between the statement coverage vectors of two test cases.
  - Estimate the number of hidden neurons and their receptive field centers
  - Compute the output of each hidden neuron

RBF Example (1)

- Suppose we have a program with ten statements. Seven test cases have been executed on the program. Table 1 gives the coverage vector and the execution result of each test.

	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	s_9	s_{10}	r
t_1	1	1	1	1	0	1	0	0	1	1	0
t_2	1	0	0	1	1	0	1	0	0	1	0
t_3	1	1	1	0	0	1	0	0	1	1	0
t_4	1	0	1	0	0	1	1	0	1	1	0
t_5	1	1	1	0	1	0	0	0	1	0	0
t_6	1	1	1	1	0	0	0	1	1	1	1
t_7	1	0	1	1	1	1	1	1	0	1	1

t_1 is a successful test

t_6 is a failed test

s_1 is executed by t_1

s_6 is not executed by t_2

RBF Example (2)

- An RBF neural network is constructed and trained
  - 10 neurons in the input layer
  - 7 neurons in the hidden layer
  - The field width σ is 0.395
  - 1 neuron in the output layer
  - The output layer weights are $\mathbf{w} = [w_1, w_2, w_3, w_4, w_5, w_6, w_7]^T$ $=[-1.326, -0.665, 0.391, -0.378, -0.308, 1.531, 1.381]^T$

RBF Example (3)

- Use the coverage vectors of the virtual test cases as the inputs to the trained network.
- The output with respect to each statement is the suspiciousness of the corresponding statement.

$$\begin{bmatrix} c_{v_1} \\ c_{v_2} \\ c_{v_3} \\ c_{v_4} \\ c_{v_5} \\ c_{v_6} \\ c_{v_7} \\ c_{v_8} \\ c_{v_9} \\ c_{v_{10}} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Part (a): Input coverage vectors

$\hat{r}_{v_1}$	0.0384	$\hat{r}_{v_6}$	0.0179
$\hat{r}_{v_2}$	0.0481	$\hat{r}_{v_7}$	0.0157
$\hat{r}_{v_3}$	0.1246	$\hat{r}_{v_8}$	0.2900
$\hat{r}_{v_4}$	0.0768	$\hat{r}_{v_9}$	0.0066
$\hat{r}_{v_5}$	0.0173	$\hat{r}_{v_{10}}$	0.0782

Highest/
Most suspicious

Lowest/
Least suspicious

Part (b): Outputs produced by the trained network which are the suspiciousness of the statements

Inputs and outputs/statement suspiciousness

BP versus RBF

- Although BP (back propagation) networks are the widely used networks for supervised learning, RBF networks (whose output layer weights are trained in a *supervised* way) are even better in our case because

RBF can learn much faster than BP networks and do not suffer from pathologies like local minima as BP networks do.

Outline

- Motivation and Background
- Execution Dice-based Fault Localization
- Suspiciousness Ranking-based Fault Localization
  - Program Spectra-based Fault Localization
  - Code Coverage-based Fault Localization
  - Statistical Analysis-based Fault Localization
  - Neural Network-based Fault Localization
  - Similarity Coefficient-based Fault Localization
- Empirical Evaluation
- Theoretical Comparison: Equivalence
- Mutation-based Automatic Bug Fixing
- Conclusions

DStar – A Similarity Coefficient-based Fault Localization

*The Construction of D** (1)

- The suspiciousness assigned to a statement should be
- Intuition 1: directly proportional to the number of failed test cases that cover it → *suspiciousness*(s) α N_{CF}
- Intuition 2: inversely proportional to the number of successful test cases that cover it → *suspiciousness*(s) α $1/N_{CS}$
- Intuition 3: inversely proportional to the number of failed test cases that do not cover it → *suspiciousness*(s) α $1/N_{UF}$
- Conveniently enough such a coefficient already exists
  Kulczynski [Kulczynski, 1928]: $N_{CF}/(N_{CS}+N_{UF})$

*The Construction of D** (*with * = 2*) (2)

- However, we also have a fourth intuition …
- Intuition 4: Intuition 1 is the most sound of the other intuitions and should therefore carry a higher weight.
- *Kulczynski* does not lead to the realization of the fourth intuition.
- Under the circumstances we might try to do something like this:

$$suspiciousness(s) = \frac{2 \times N_{CF}}{N_{UF} + N_{CS}}$$ or maybe even $$suspiciousness(s) = \frac{100 \times N_{CF}}{N_{UF} + N_{CS}}$$

- But this is not going to help us (as we shall later see)
- So instead we make use of a different coefficient (D*)

$$suspiciousness(s) = \frac{N_{CF} \times N_{CF}}{N_{UF} + N_{CS}}$$

Back to Slide 118

D* Example : with * = 2 (1)

- Suppose we are writing a program that computes the sum or average of two numbers.
  - But with respect to the sum computation (statement 5), instead of adding the two numbers, we accidentally subtract them

Stmt. #.	Program (*P*)	Coverage t_1	t_2	t_3	t_4	t_5	t_6
1	read (a);	•	•	•	•	•	•
2	read (b);	•	•	•	•	•	•
3	read (choice);	•	•	•	•	•	•
4	if (choice == "sum")	•	•	•	•	•	•
5	result = a - b; //Correct: a + b;	•	•	•			
6	else if (choice == "average")				•	•	•
7	result = (a + b) / 2;				•	•	•
8	print (result);	•	•	•	•	•	•
Execution Result (0 = Successful / 1 = Failed)		1	1	0	0	0	0

D* Example: with * = 2 (2)

- Next we collect the statistics we need for D* (N_{CF}, N_{UF} and N_{CS})

Stmt. #	N_{CF}	N_{UF}	N_{CS}	Suspiciousness based on D* $N_{CF} \times N_{CF} / (N_{UF}+N_{CS})$
1	2	0	4	1
2	2	0	4	1
3	2	0	4	1
4	2	0	4	1
5	2	0	1	4
6	0	2	3	0
7	0	2	3	0
8	2	0	4	1

Most suspicious

Statement ranking: **5**, 1, 2, 3, 4, 8, 6, 7

Tied together

Tied together

Other Fault Localization Techniques

Tarantula, Ochiai, SOBER, & Liblit05

- Tarantula

$$suspiciousness(e) = \frac{\dfrac{failed(e)}{totalfailed}}{\dfrac{passed(e)}{totalpassed} + \dfrac{failed(e)}{totalfailed}}$$

  - *passed(e)* is the number of passed test cases that execute statement *e* one or more times
  - *failed(e)* is the number of failed test cases that execute statement *e* one or more times
  - *totalpassed* is the total number of test cases that pass in the test suite
  - *totalfailed* is the total number of test cases that fail in the test suite

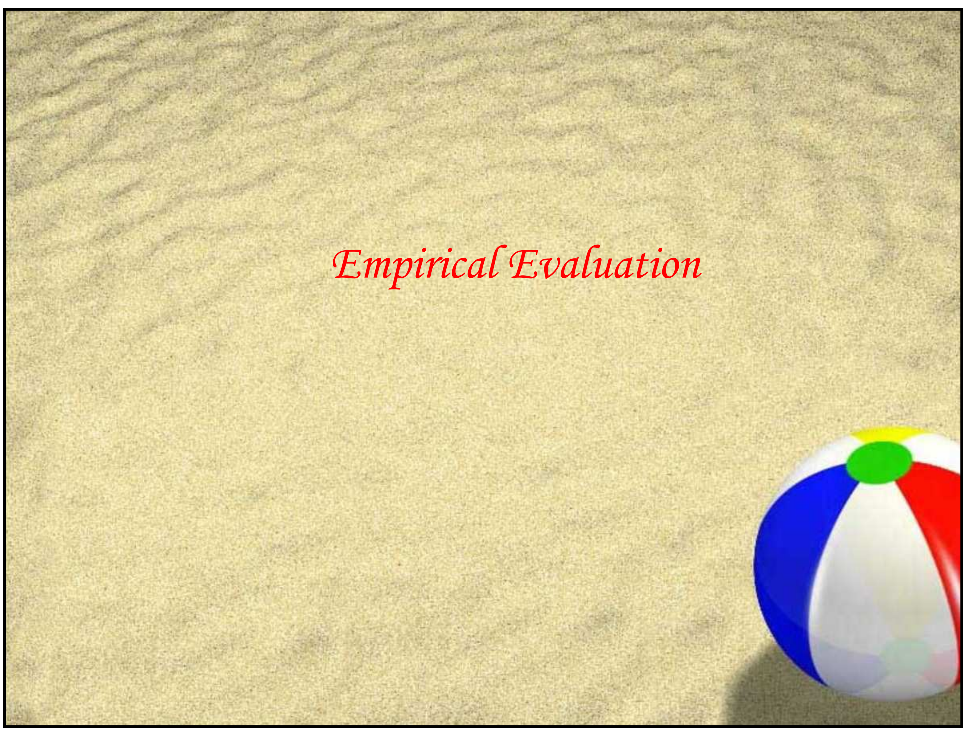
- Ochiai

$$\frac{N_{CF}}{\sqrt{N_F \times (N_{CF} + N_{CS})}}$$

- SOBER
- Liblit05

Outline

- Motivation and Background
- Execution Dice-based Fault Localization
- Suspiciousness Ranking-based Fault Localization
  - Program Spectra-based Fault Localization
  - Code Coverage-based Fault Localization
  - Statistical Analysis-based Fault Localization
  - Neural Network-based Fault Localization
  - Similarity Coefficient-based Fault Localization
- Empirical Evaluation
- Theoretical Comparison: Equivalence
- Mutation-based Automatic Bug Fixing
- Conclusions



Is a Technique Good at Locating Faults?

- "Good" is more of a relative term. We can show a fault localization technique is good by showing that it is more effective than other competing techniques
- We do this via rigorous case studies
  - Using a comprehensive set of subject programs
  - Comparing the effectiveness between different fault localization techniques
  - Evaluating across multiple criteria
- Since it is not possible to theoretically prove that one fault localization technique is always more effective than another, such empirical evaluation is typically the norm
  - We will return to this issue later on

Jump to Slide 123

Subject Programs

- Four sets of subject programs – the *Siemens* suite, the *Unix* suite, *gzip* and *Ant* – were used (19 different programs in all – C & Java)
  - Two additional programs (*grep* and *make*) are also used which makes a total of 21 programs

Program	Lines of Code	Number of faulty versions used†	Number of test cases
print_tokens	565	5	4130
print_tokens2	510	10	4115
schedule	412	9	2650
schedule2	307	9	2710
replace	563	32	5542
tcas	173	41	1608
tot_info	406	23	1052
cal	202	20	162
checkeq	102	20	166
col	308	30	156
comm	167	12	186
crypt	134	14	156
look	170	14	193
sort	913	21	997
spline	338	13	700
tr	137	11	870
uniq	143	17	431
gzip	6573	28	211
Ant	75333	23	871

† Some versions were created using mutation-based fault injection

Techniques D* is Compared to

- First compared D* to the *Kulcyznski* coefficient

- Also compared it with 11 other well-known coefficients forming a baker's dozen [Choi et al. 2010, Willett 2003]

(1) *Simple-Matching*
(2) *BraunBanquet*
(3) *Dennis*
(4) *Mountford*
(5) *Fossum*
(6) *Pearson* (χ^2)
(7) *Gower*
(8) *Michael*
(9) *Pierce*
(10) *Baroni-Urbani/Buser*
(11) *Tarwid*

- Further comparisons with other techniques were also performed
  - To be discussed later *i*

Three Evaluation Metrics/Criteria

- Number of statements examined
  - The number of statements that need to be examined by D* to locate faults versus other techniques
  - An absolute measure

- The *EXAM* score: the percentage of code examined
  - The percentage of code that needs to be examined by using D* to locate faults versus other techniques
  - A relative (graphical) measure

- The Wilcoxon Signed-Rank Test
  - Evaluate the alternative hypothesis that other techniques will require the examination of *more statements than D**
    - D* is more effective than other techniques
    - Null hypothesis being that the other techniques require the examination of a number of statements that is *less than or equal to* that required by D*
  - A statistical measure

Ties in the Ranking: Best/Worst

- The suspiciousness assigned to a statement by D* (and other techniques) may not be unique, i.e., two or more statements can be tied for the same position in the ranking.

From our example:

Statement ranking: **5**, $\boxed{1, 2, 3, 4, 8}$, 6, 7

Tied together — Tied together

- Assuming a faulty statement and some correct statements are tied
  - In the *best* case we examine the faulty statement *first*
  - In the *worst* case we examine it *last*

- For each of the previously discussed evaluation criteria, we will have the ***best case*** and the ***worst case*** effectiveness.
  - Presenting only the ***average*** would have resulted in a loss of information

Results – Total Number of Statements Examined

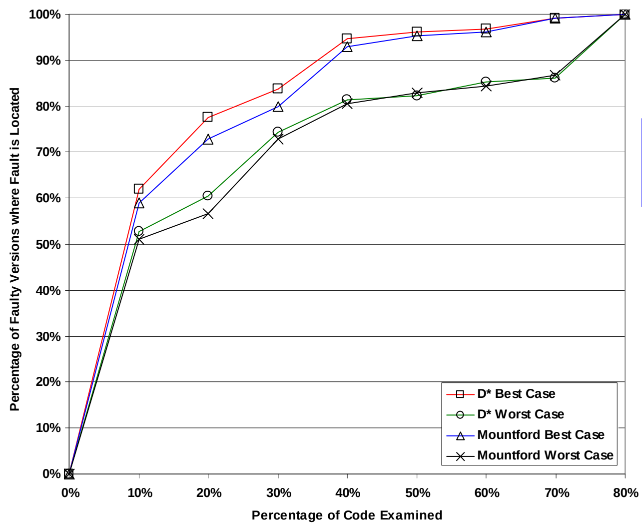
Fault Localization Technique	Best Case				Worst Case			
	Siemens	Unix	gzip	Ant	Siemens	Unix	gzip	Ant
D*	**1754**	**1805**	**1220**	**672**	**2650**	**5226**	**3087**	**1184**
Kulcynzki	2327	2358	1272	1557	3186	5779	3139	2069
Simple-Matching	6335	5545	9087	250414	7187	8977	10968	253631
BraunBanquet	2438	2767	1358	2196	3296	6187	3135	2698
Dennis	2206	2934	1960	1974	3074	6504	3737	2476
Mountford	1974	2183	1317	3298	2832	5644	3111	3818
Fossum	2230	2468	4547	150415	3126	5843	8701	150917
Pearson	3279	3581	1450	1188	4247	7221	3227	1690
Gower	6586	8630	26215	967307	7434	12027	27992	967809
Michael	1993	3713	2504	4502	2864	7283	4281	5004
Pierce	8072	11782	24065	322033	15299	23387	46753	1018725
Baroni-Urbani/Buser	3547	3189	1428	4693	4404	6605	3205	5195
Tarwid	2453	3399	3110	5964	3321	7883	5032	9935

D* is clearly the most effective

Jump to Slide 119

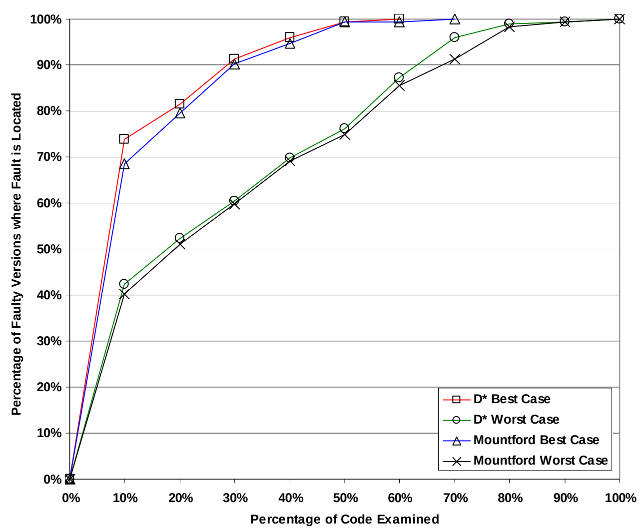
- D^* is very consistent in its performance
- Often the worst case of D^* is better than the best case of the other techniques (Note that * = 2)

Results – EXAM Score (Siemens suite)



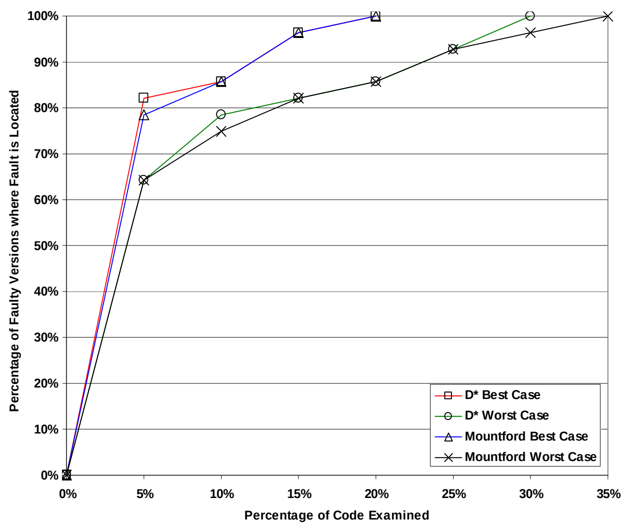
D* is clearly the most effective

Results – EXAM Score (Unix suite)



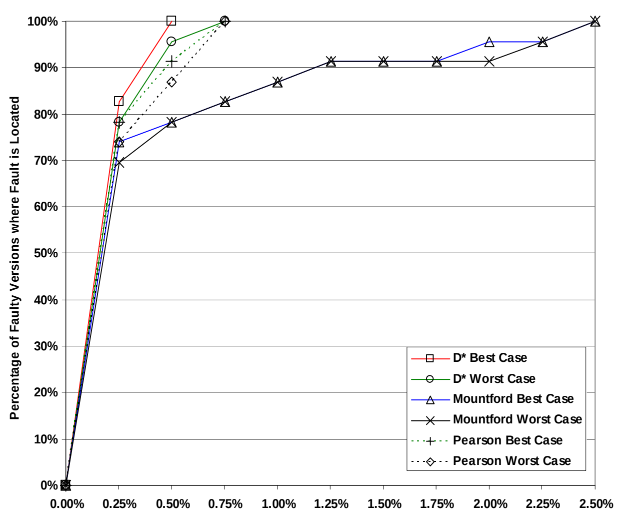
D* is clearly the most effective

Results – EXAM Score (Gzip)



D* is clearly the most effective

Results – EXAM Score (Ant)



D* is clearly the most effective

Results – Wilcoxon Signed-Rank Test (1)

Fault Localization Technique	Best Case				Worst Case			
	Siemens	Unix	gzip	Ant	Siemens	Unix	gzip	Ant
Kulcynzki	99.99%	99.99%	93.75%	98.43%	99.99%	99.99%	93.75%	98.43%
Simple-Matching	100%	100%	99.80%	99.90%	100%	100%	97.60%	99.80%
BraunBanquet	99.99%	100%	99.80%	99.80%	99.99%	99.99%	71.43%	99.21%
Dennis	99.99%	100%	99.99%	99.80%	99.99%	100%	94.20%	99.21%
Mountford	99.99%	99.99%	99.21%	99.90%	99.99%	99.99%	73.82%	99.80%
Fossum	100%	99.99%	99.21%	99.21%	100%	99.99%	99.62%	96.87%
Pearson	100%	99.99%	99.21%	99.21%	100%	99.99%	70.87%	96.87%
Gower	100%	100%	99.99%	99.99%	100%	100%	99.99%	99.99%
Michael	99.68%	99.99%	99.99%	99.97%	99.54%	99.99%	99.99%	99.97%
Pierce	100%	100%	99.99%	99.99%	100%	100%	99.99%	99.99%
Baroni-Urbani/Buser	99.99%	100%	99.80%	99.80%	99.99%	100%	74.42%	98.82%
Tarwid	99.99%	99.99%	99.99%	99.99%	99.99%	100%	99.99%	99.99%

- Generally the confidence with which we can claim that D* is more effective than the other techniques is very high (easily over 99%).
- But there are a few exceptions.
- Why? Perhaps this has something to do with the way our hypothesis was constructed.

Results – Wilcoxon Signed-Rank Test (2)

- Let us modify our alternative hypothesis to consider *equalities*.
  - We now evaluate to see if D* is more effective than, or at least as effective as, the other techniques.
  - Which is to say D* requires the examination of a number of statements that is less than or equal to that required by the other techniques.

Fault Localization Technique	Best Case		Worst Case	
	gzip	Ant	gzip	Ant
Kulcynzki	100%	100%	100%	100%
Simple-Matching	100%	100%	99.94%	99.90%
BraunBanquet	100%	100%	99.14%	99.61%
Dennis	100%	100%	99.43%	99.61%
Mountford	100%	100%	95.78%	99.90%
Fossum	100%	100%	99.67%	99.44%
Pearson	100%	100%	92.19%	98.44%
Baroni-Urbani/Buser	100%	100%	95.42%	99.22%

D* is clearly the most effective

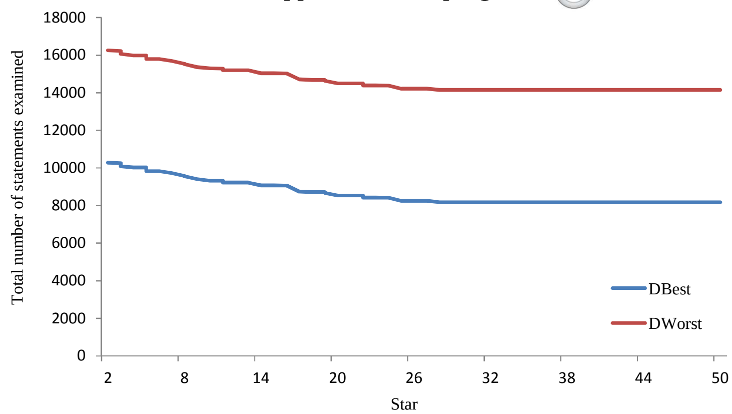
Confidence levels have gone up significantly. All entries but one are greater than 95%.

More Discussion on D*

- D* with a higher value for the *
- Compare D* with other fault localization techniques

Effectiveness of D*

- The effectiveness of D* for the *make* program increases until it levels off as the value of * increases.
- A similar observation also applies to other programs. (i)



Effectiveness of Other Fault Localization Techniques

- The best- and worst-case effectiveness of 18 fault localization techniques (excluding D*) on 21 different programs.

	Best Case						Worst Case					
	Unix	Simens	grep	gzip	make	Ant	Unix	Simens	grep	gzip	make	Ant
H3c	1655	1396	2702	1535	8553	1320	5026	2292	4435	3312	14272	1882
H3b	1701	1439	3019	1535	10817	1358	5072	2335	4752	3313	16556	1860
RBF	1302	2114	2075	2966	9188	233	4758	2980	3964	4743	14590	759
Ochiai	1906	1796	3092	1270	10305	887	5322	2692	4825	3047	16044	1389
Crosstab	2524	2005	4005	1314	12403	1076	6094	2873	7443	3091	18142	1578
Tarantula	3394	2453	5793	3110	16890	5964	7704	3311	7812	5032	23468	9935
Kulcynzki	2358	2327	3458	1272	10701	1557	5779	3186	5192	3139	16668	2069
Simple-Matching	5545	6335	23806	9087	41374	250414	8977	7187	25606	10968	48401	253631
BraunBanquet	2767	2438	4114	1358	11734	2196	3296	3296	5847	3135	17986	2698
Dennis	2934	2206	5498	1960	15016	1974	6504	3074	8936	3737	20755	2476
Mountford	2183	1974	3450	1317	11269	3298	5644	2832	5189	3111	17152	3818
Fossum	2468	2230	15952	4547	19567	150415	5843	3126	21193	8701	25036	150917
Pearson	3581	3279	6894	1450	17689	1188	7221	4247	10796	3227	23569	1690
Gower	8630	6586	43428	26215	128318	967307	12027	7434	45262	27992	134057	967809
Michael	3713	1993	5027	2504	14986	4502	7283	2864	8501	4281	20725	5004
Pierce	11782	8072	16646	24065	30568	322033	23387	15299	60437	46753	164856	1018725
Baroni-Urbani/Buser	3189	3547	4902	1428	12130	4693	6605	4404	6635	3205	17689	5195
Tarwid	3399	2453	5793	3110	16890	5964	7883	3321	9517	5032	23468	9935

Jump to Slide 105

Comparison between D and Other Techniques*

- The effectiveness of D^2 is better than the other 12 similarity coefficient-based fault localization techniques. *i*

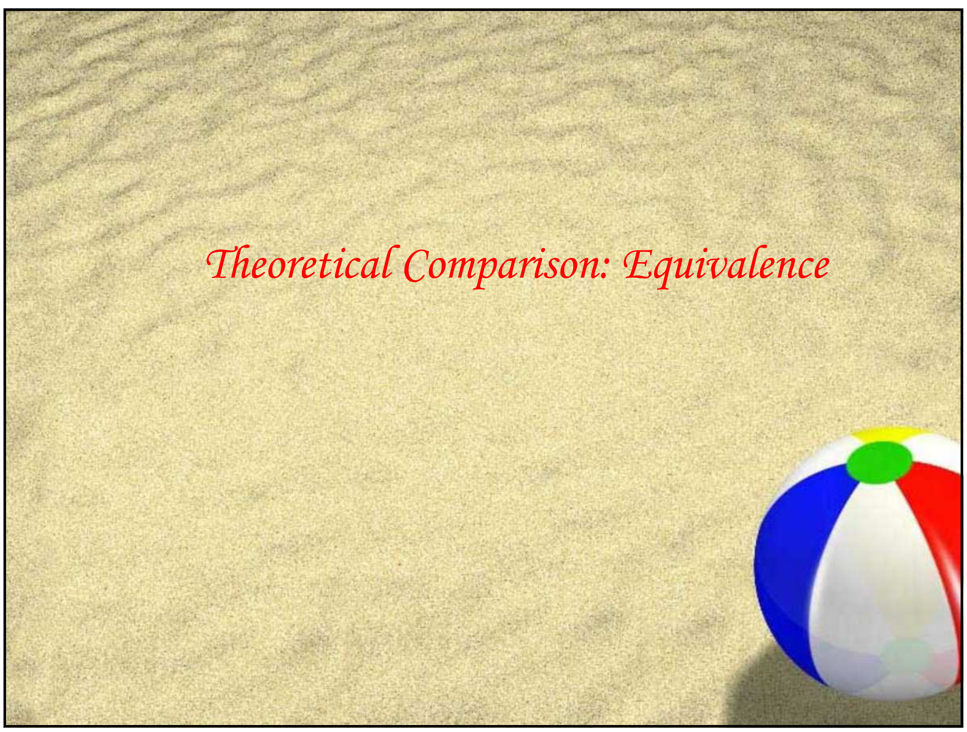
- From the following table, we also observe that D* (*with an appropriate value of **) performs better than other fault localization techniques, regardless of the subject programs, and the best- or worst-case.
  - The cell with a black background gives the smallest * such that D* outperforms others.

	Best Case						Worst Case					
	Unix	Simens	grep	gzip	make	Ant	Unix	Simens	grep	gzip	make	Ant
D^2	1805	1754	3023	1220	10287	672	5226	2650	4757	3087	16254	1184
D^3	1667	1526	2946	1088	10257	368	5088	2422	4680	2955	16224	880
D^4	1594	1460	2833	1087	10022	293	5015	2356	4567	2954	15989	805
D^5	1507	1435	2762	1085	10022	228	4928	2331	4496	2952	15989	740
D*		1386 (*=7)	2693 (*=8)		8529 (*=20)			2284 (*=7)	4427 (*=8)		14219 (*=25)	
H3b	1701	1439	3019	1535	10817	1358	5072	2335	4752	3313	16556	1860
H3c	1655	1396	2702	1535	8553	1320	5026	2292	4435	3312	14272	1882
Tarantula	3394	2453	5793	3110	16890	5964	7704	3311	7812	5032	23468	9935
Ochiai	1906	1796	3092	1270	10305	887	5322	2692	4825	3047	16044	1389

Jump to Slide 106

Outline

- Motivation and Background
- Execution Dice-based Fault Localization
- Suspiciousness Ranking-based Fault Localization
  - Program Spectra-based Fault Localization
  - Code Coverage-based Fault Localization
  - Statistical Analysis-based Fault Localization
  - Neural Network-based Fault Localization
  - Similarity Coefficient-based Fault Localization
- Empirical Evaluation
- Theoretical Comparison: Equivalence
- Mutation-based Automatic Bug Fixing
- Conclusions



Comparing Fault Localization Techniques (1)

- As discussed earlier the general norm for comparing fault localization techniques has been to use *empirical data.*
- If technique α is better than technique β, then it should lead programmers to the location of fault(s) faster than β.
- Multiple metrics have been proposed to do this such as the ones used in our research. ⓘ
- Case studies can be quite expensive and time-consuming to perform. Often a lot of data has to be analyzed.

But is empirical comparison always required…especially when trying to show that two techniques will be equally effective?

Comparing Fault Localization Techniques (2)

- Note that the suspiciousness of a statement is irrelevant from an absolute sense.
  - It only matters how the suspiciousness of two (or more) statements compare with respect to each other (i.e., relative to one another).
- Supposing we have two statements s_1 and s_2 with suspiciousness values of 5 and 6, respectively. This means that s_2 is ranked above s_1 as it is more suspicious.
- However, s_2 would still be ranked above s_1 if the suspiciousness values were 6 and 7, or 50 and 60, respectively – the relative ordering of s_1 and s_2 is still maintained.
- Thus, subtracting the same constant from (or adding it to) the suspiciousness of every statement will have no effect on the final ranking. The same applies for multiplication/division operations.

Comparing Fault Localization Techniques (3)

- Recall the suspiciousness computation of *Kulczynski*

$$suspiciousness(s) = \frac{N_{CF}}{N_{UF} + N_{CS}}$$

- It now becomes clear that an identical ranking will be produced by

$$suspiciousness(s) = \left(\frac{N_{CF}}{N_{UF} + N_{CS}}\right) + 1 \quad \text{or} \quad suspiciousness(s) = \left(\frac{N_{CF}}{N_{UF} + N_{CS}}\right) \times 10$$

- This is why D* was constructed the way it was

- Any operation that is *order-preserving* can be safely performed on the suspiciousness function without changing the ranking.

- If the ranking does not change…then the effectiveness will not change either. ***We can exploit this!***

Comparing Fault Localization Techniques (4)

- Consider a program *P* with a set of elements $\mathcal{M}$. Let *rank*(*r*,*s*) be a function that returns the position of statement *s* in ranking *r*.

- Two rankings r_α and r_β (produced by using two techniques $\mathcal{L}_\alpha$ and $\mathcal{L}_\beta$ on the same input data) are *equal* if
  - $\forall s \in \mathcal{M}, rank(r_\alpha, s) = rank(r_\beta, s)$.
  - Two rankings are equal if for every statement, the position is the same in both rankings.

- If two fault localization techniques $\mathcal{L}_\alpha$ and $\mathcal{L}_\beta$ always produce rankings that are equal, then the techniques are said to be equivalent, i.e., $\mathcal{L}_\alpha \equiv \mathcal{L}_\beta$ and therefore will always be equally as effective (at fault localization).

- **So is the equivalence relation useful?**

  Certainly! In at least two scenarios it holds great potential
  - Eliminating the need for time-consuming case studies.
  - Making suspiciousness computations more efficient.

Eliminating the Need for Case Studies (1)

- Take the example of [Abreu et al. 2009] where
  - The authors use of the *Ochiai* coefficient to compute suspiciousness.
  - The coefficient is compared to several other coefficients *empirically*.
  - Among others, it is compared to the *Jaccard* and *Sorensen-Dice* coefficients.
- We posit that this was unnecessary, as per the equivalence relation.

Jaccard

$$suspiciousness(s) = \frac{N_{CF}}{N_{CF} + N_{UF} + N_{CS}}$$

Sorensen-Dice

$$suspiciousness(s) = \frac{2N_{CF}}{2N_{CF} + N_{UF} + N_{CS}}$$

- Via a set of order-preserving operations, both can be reduced to: $suspiciousness(s) = \boxed{\frac{N_{CF}}{N_{UF} + N_{CS}}}$

Jaccard ≡ Sorensen-Dice

R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. C. van Gemund, "A Practical Evaluation of Spectrum-based Fault Localization," Journal of Systems and Software, 82(11):1780-1792, November 2009

Eliminating the Need for Case Studies (2)

- As it turns out the coefficient *Anderberg* also evaluates to the same form. Ochiai was empirically compared to Anderberg.

Jaccard ≡ Sorensen-Dice ≡ Anderberg

- In fact the authors also compared Ochiai to the *SimpleMatching* and *Rogers and Tanimoto* coefficients, the both of which are also equivalent to one another.

SimpleMatching ≡ Rogers and Tanimoto

Such redundant comparisons could have been avoided by making use of the fault localization equivalence relation.

Making Computations More Efficient (1)

- As shown, if Jaccard were the chosen fault localization technique, using the suspiciousness function

$$suspiciousness(s) = \frac{N_{CF}}{N_{CF} + N_{UF} + N_{CS}}$$

would give the same results as using

$$suspiciousness(s) = \frac{N_{CF}}{N_{UF} + N_{CS}}$$

- We should go with the simplest computation as it is expected to be faster.

Making Computations More Efficient (2)

- We performed an additional case study on the 7 programs of the Siemens suite
- Observed the relative time saved in computing suspiciousness for all the statements in a faulty program, by using the *simplified form* of Jaccard (*J**) as opposed to the *original* (*J*).
  - The quantity (*J*–*J**) represents the computational time that is saved.
  - ((*J*–*J**)/*J*)×100% represents the relative time saved, i.e., efficiency gained.
- 100 trials were performed per faulty version.
- Difference in times was computed to nanosecond precision.

Making Computations More Efficient (3)

Programs	Average Percentage Time Saved
print_tokens	35.37%
print_tokens2	39.21%
schedule	44.62%
schedule2	49.74%
replace	41.65%
tcas	52.46%
tot_info	47.68%

- The savings in terms of time are quite significant.

- Using the equivalence relation can thus, help reduce techniques to simplified forms, thereby greatly increasing efficiency.

Programs with Multiple Faults

- One bug at a time
- A good approach is to use *"fault-focused" clustering.*
  - Divide failed test cases into clusters that target different faults
  - Failed test cases in each fault-focused cluster are combined with the successful tests for debugging a single fault.

Outline

- Motivation and Background
- Execution Dice-based Fault Localization
- Suspiciousness Ranking-based Fault Localization
  - Program Spectra-based Fault Localization
  - Code Coverage-based Fault Localization
  - Statistical Analysis-based Fault Localization
  - Neural Network-based Fault Localization
  - Similarity Coefficient-based Fault Localization
- Empirical Evaluation
- Theoretical Comparison: Equivalence
- Mutation-based Automatic Bug Fixing
- Conclusions

Mutation-based Automatic Bug Fixing

V. Debroy and W. E. Wong, "Using Mutation to Automatically Suggest Fixes for Faulty Programs," in *Proceedings of the 3rd International Conference on Software Testing, Verification and Validation* (ICST), Paris, France, April 2010

Mutation as a Fault Generation Aid

- For research experiments, large comprehensive data sets are rarely available
- Need faulty versions of programs to perform all kinds of experiments on, but don't always have a way to get them
- Recently many researchers have relied on mutation
  - Mutants generated can represent realistic faults
  - Experiments that use these mutants as faulty versions can yield trustworthy results
  - As opposed to seeding faults, mutant generation is automatic

Jump to Slide 105

Mutation as a Fault Fixing Aid?

If mutating a correct program can produce a realistic fault, can mutating an incorrect program produce a realistic fix?

- Supposing we wanted to write program *P*
- But we ended up writing a faulty program *P'*
  - We know *P'* is faulty because at least one test case in our test set results in failure when executed on *P'*
- Mutate *P'* to get *P"*
- If *P"* = *P*… we automatically fixed the fault in *P'*

Our Solution

Mutation	***Fault Localization***
The Good: Can result in potential fixes for faulty programs automatically.	**The Good:** Can potentially identify the location of a fault in a program.
The Bad: We have no idea as to where in a program a fault is, and so we do not know how to proceed. Randomly examining mutants can be prohibitively expensive.	**The Bad:** Even if we locate the fault, we have no idea as to how to fix the fault. This is left solely as the responsibility of the programmers/debuggers.

So…what if we combined the two?

Conclusion

What We Have Discussed

- **Existing and new fault localization techniques**
  - Many of them use the same information (statement coverage and execution results) to identify suspicious code likely to contain program bug(s)

- **A strategy to automatically suggest fixes for faults** that
  - makes as few assumptions as possible about the software being debugged
  - is generally applicable to different types of software and programming languages
  - still manages to produce some useful information even when it is unable to fix faults automatically

Present a framework to automate the debugging process.