Survey of Software Bug Localization Using Dynamic Analysis

DR. Hua Jie Lee (Jason)

Senior Software Test Engineer
DOLBY LABORATORIES, Australia

February 21, 2013

Outline

Common Terms Used I

1. Statement – program statements
2. Predicate – specific program point such as conditions of if-statements, return expression of functions etc.
3. Metric – numeric function and as a standard of measurement. Not related to metric spaces (where it defines the distance between any two points in the space)
4. Rank percentages – percentages of the program code need/need not be examined to locate software bug

Common Terms Used II

1. Bug – code segment of program code which causes failure of the program
2. Single bug – also known as one-bug
3. Fault – incorrect statement or data definition in a program
4. Failure – deviation of the observed behaviour of a program, or a system, from its specification

Common Terms Used III

1. Binary weighting approach – using program spectra information without introducing any weights
2. $e_{s,t}$ – test execution coverage of statement s for a respective test t
3. DU-PAIR refers to associating a conditional branch with statements that exist before the conditional statement or by the conditional statement itself

Common Terms Used IV

1. Graph – collection of vertices and edges
2. Vertices – better known as nodes
3. Edges – connector of pairs of vertices

Introduction

- Software bugs cost U.S economy USD 59.5 billion dollars (0.6% of GDP) in 2002 [Hailpern et al. '02].
- Dynamic analysis - one of the software bug localization techniques which could improve efficiency in locating bugs.

Software Bug Incidents I

YEAR	COMPANY	OUTCOME
2013	Network manufacturers	40–50 million devices vulnerable to the attack from bugs found in networking equipment (UPnP standard)
2012	Knight Capital	Computer bug costs the loss of USD 440 million where a series of automatic order of company's fund were executed instead of over period of days

Software Bug Incidents II

YEAR	COMPANY	OUTCOME
2011	Apple	Alarm not working on the 1 Jan 2011. Caused customers missed flights and to work. Suspected to be the bug in the complexity of the algorithm of week-number
2010	NAB	NAB system crashed. Delayed payments to customers. Due to the bug in the batch processing software code that contained instructions on how the system operate in the batch processing operations

Software Bug Incidents III

YEAR	COMPANY	OUTCOME
1990	AT & T Wireless	Massive shutdown of their network [Neumann, 1990] due to a single line of code (bug) which is part of software upgrade to speed up calling
1962	NASA	Incorrect signals guidance (due to a missing superscript bar in a written formula) causes the launch of first Mariner mission Mariner I rocket to off course

And the list goes on, and on ...

Background

1. Static analysis [Weiser et al. ’79, Weiser et al. 81, Weiser et al. 82].
2. Dynamic analysis
   - Test coverage information approaches [Jones et al. ’02, Jones et al. ’05, Abreu et al. ’06, Abreu et al. ’07, Wong et al. ’07, Naish et al. ’09, Hua Jie, L. et al. ’10]
   - Slicing approaches [Korel, B. et al. ’90, Agrawal, H. et al. ’90, Korel, B. et al. ’97]
   - Statistical approaches [Liblit et al. ’05, Liu et al. ’05, Jiang et al. ’05]
   - State-based Approaches[Zeller ’01, Zeller et al. ’05]
   - Machine learning approaches [Dickinson et al. ’01, Jiang et al. ’05, Liblit et al. ’05, Fatta et al. ’06]

What is Program Spectra I?

1. A type of dynamic analysis approach
2. A path (or program) spectrum is a collection of data that provides a specific view of the dynamic behavior of software.
3. Contains information about which part of a program (individual statements, basic blocks or functions) was executed during the execution of several test cases.

What is Program Spectra II?

1. a_{np} – number of pass test cases not executing respective statements
2. a_{nf} – number of fail test cases not executing respective statements
3. a_{ep} – number of pass test cases executing respective statements
4. a_{ef} – number of fail test cases executing respective statements
5. $totF$ – total number of fail test cases
6. $totP$ – total number of pass test cases

Example of test coverage information (frequency counts) with tests $T_1 \ldots T_5$

	T_1	T_2	T_3	T_4	T_5
S_1	60	2	100	50	38
S_2	70	65	90	52	45
S_3	25	35	0	0	42
S_4	59	0	37	0	4
S_5	0	80	0	0	57
Test Result	Fail	Fail	Fail	Pass	Pass

Example of test coverage information (frequency counts) with tests $T_1 \ldots T_5$

	T_1	T_2	T_3	T_4	T_5
S_1	60	2	100	50	38
S_2	70	65	90	52	45
S_3	25	35	0	0	42
S_4	59	0	37	0	4
S_5	0	80	0	0	57
Test Result	Fail	Fail	Fail	Pass	Pass

Example of test coverage information (binary) and program spectra with tests $T_1 \ldots T_5$

	T_1	T_2	T_3	T_4	T_5	a_{np}	a_{nf}	a_{ep}	a_{ef}
S_1	1	1	1	1	1	0	0	2	3
S_2	1	1	1	1	1	0	0	2	3
S_3	1	1	0	0	1	1	1	1	2
S_4	1	0	1	0	1	1	1	1	2
S_5	0	1	0	0	1	1	2	1	1
⋮									
Test Result	Fail	Fail	Fail	Pass	Pass				

Definition of various ranking metrics used I

Numerous metrics proposed to predict which statements are most likely to be buggy.

- Software automated debugging tools (Tarantula (Tar), Ample, Wong3, Wong4, Zoltar, CBI Inc),
- Self-organizing-map (Hamann, Kulczynski2 (Kul2)),
- Zoology (Ochiai), and
- Botany (Jaccard, Sorensen-Dice(Sor), Anderberg, Rogers and Tanimoto (Rog), Russell and Rao(Rus), Simple-Matching, Ochiai2).
- Most spectra metrics formula used primarily on a_{ef} , secondarily on a_{ep}.

▸ More Spectra Metrics

Benchmarks

Description of Siemens Test Suite, Unix, Concordance and Space

Program	Ver	1 Bug	2 Bugs	3 Bugs	LOC	Test Cases
tcas	41	37	604	—	173	1608
schedule	9	8	—	—	410	2650
schedule2	10	9	28	131	307	2710
print_tokens	7	6	—	—	563	4130
print_tokens2	10	10	10	44	508	4115
tot_info	23	23	245	782	406	1052
replace	32	29	34	16	563	5542
Col	28	28	147	2030	308	156
Cal	18	18	115	1475	202	162
Uniq	14	14	14	114	143	431
Spline	13	13	20	174	338	700
Checkeq	18	18	56	502	102	332
Tr	11	11	17	90	137	870
Concordance	13	11	—	—	1492	372
Space	38	15	5	1	9059	13585

How to measure the performance of locating bugs?

	T_1	T_2	T_3	T_4	T_5	a_{np}	a_{nf}	a_{ep}	a_{ef}
S_1	1	1	1	1	1	0	0	2	3
S_2	1	1	1	1	1	0	0	2	3
S_3	1	1	0	0	1	1	1	1	2
S_4	1	0	1	0	1	1	1	1	2
S_5	0	1	0	0	1	1	2	1	1
Test Result	Fail	Fail	Fail	Pass	Pass				

How to measure the performance of locating bugs?

	T_1	T_2	T_3	T_4	T_5	a_{np}	a_{nf}	a_{ep}	a_{ef}
S_1	1	1	1	1	1	0	0	2	3
S_2	1	1	1	1	1	0	0	2	3
S_3	1	1	0	0	1	1	1	1	2
S_4	1	0	1	0	1	1	1	1	2
S_5	0	1	0	0	1	1	2	1	1
Test Result	Fail	Fail	Fail	Pass	Pass				

Definition of Tarantula metric

Name	Formula
Tarantula	$\frac{\frac{a_{ef}}{a_{ef}+a_{nf}}}{\frac{a_{ef}}{a_{ef}+a_{nf}}+\frac{a_{ep}}{a_{ep}+a_{np}}}$

How to measure the performance of locating bugs?

	T_1	T_2	T_3	T_4	T_5	a_{np}	a_{nf}	a_{ep}	a_{ef}
S_1	1	1	1	1	1	0	0	2	3
S_2	1	1	1	1	1	0	0	2	3
S_3	1	1	0	0	1	1	1	1	2
S_4	1	0	1	0	1	1	1	1	2
S_5	0	1	0	0	1	1	2	1	1
Test Result	Fail	Fail	Fail	Pass	Pass				

Definition of Tarantula metric

Name	Formula
Tarantula	$\frac{\frac{a_{ef}}{a_{ef}+a_{nf}}}{\frac{a_{ef}}{a_{ef}+a_{nf}}+\frac{a_{ep}}{a_{ep}+a_{np}}}$

- Let S_1 as the bug.
- Tarantula (S_1=0.5,S_2=0.5,S_3=0.5714,S_4=0.5714,S_5=0.4000)
- Rank percentages has been used - 10% refers to 10% of the statements in the program must be inspected before the bug is located.
- Ties handled differently - **Best case**= $\frac{3}{5} * 100\%$. **Worst case**= $\frac{4}{5} * 100\%$. **Average case**=$\frac{3.5}{5} * 100\%$.

Jones et al. Proposed Approach I - Discrete [Jones et al., 2002]

1. Two approaches to distinguish buggy statements
2. *discrete* and *continuous approach*

Discrete Approach

1. Uses three-color system (Red, Green and Yellow)
2. Statements executed by pass test case(s) visualized Green
3. Statements executed by fail test case(s) visualized Red
4. Statements executed by BOTH pass and fail test case(s) visualized Yellow
5. Unable to distinguish bug if buggy statement executed by both pass and fail test cases

Jones et al. Proposed Approach II - Continuous

Definition (Color Visualization using Continuous Approach for a statement *s*)

$$color = low\ color\ (red) + \left(\frac{\frac{a_{ep}}{totP} * 100\%}{\frac{a_{ep}}{totP} * 100\% + \frac{a_{ef}}{totF} * 100\%}\right) * color\ range$$

1. Map color for statements
2. *low color* is defined as Red for one of the color spectrum
3. *color range* defined as the other end of spectrum (0 - Red, 100 - Green)
4. Statement that is executed most of the time either by pass/fail test cases assigned a brighter color.

Jones et al. Proposed Approach III

Definition (Brightness for a statement s)

$$bright = max(\frac{a_{ep}}{totP} * 100\%, \frac{a_{ef}}{totF} * 100\%)$$

Definition (Tarantula metric)

$$\frac{\frac{a_{ef}}{a_{ef}+a_{nf}}}{\frac{a_{ef}}{a_{ef}+a_{nf}} + \frac{a_{ep}}{a_{ep}+a_{np}}}$$

Jones et al. Proposed Approach IV

1. Define brightness component in the scale of 0 to 100
2. Statement executed less by test cases assigned darker color and vice versa
3. Gimp [GIMP] is used to define values of colors
4. Color for statement *s* would be a single number representing color spectrum

Jones et al. Proposed Approach V

1. More pass test cases execute program statement, color mapped towards Green spectrum
2. More fail test cases execute program statement, color mapped towards Red spectrum
3. Same number of pass and fail test cases execute program statement, color mapped towards Yellow spectrum
4. Color could distinguish buggy from non-buggy statements
5. Space [Do et al., 2005] is used as benchmark

Jones et al. Proposed Approach VI

1. Tarantula could not distinguish the multiple bugs in the program by judging the color spectrum; only single bug programs
2. Evaluated with other approaches and outperformed others; Intersection model, Union model, Nearest Neighbor model [Renieres et al., 2003] and Cause Transitions [Cleve et al., 2005]

Wong et al. Proposed Approach I [Wong et al., 2007]

1. Proposed several spectra metrics; Wong1, Wong2 and Wong3
2. Less weight given to pass test cases if particular statement executed by more pass test cases
3. χsuds [Telecordia, 1998] is used to instrument and perform test executions on the program

Wong et al. Proposed Approach II

1. Wong1 - depends on the number of fail test cases.
2. Wong2 - depends on both the number of pass and fail test cases.
3. Wong3 - used a constant value, $\alpha = 0.001$ to adjust weights for test cases
4. Used Siemens Test Suite as benchmark
5. Proposed another metric, Wong4 [Wong et al., 2009], adjust both pass and fail test cases and $\alpha = 0.0001$

Naish et al. Proposed Approach I [Naish et al., 2009a]

1. Proposed optimal metrics, O and O^p for locating single-bug programs using statement-based spectra.
2. Used If-Then-Else (ITE2) model program to understand single bug programs
3. Model program can be used to generate and simulate different program models (if-then-else, nested conditional expression)

$ITE2_8$ Model Program

```
if (t1())
    s1();          /* S1 */
else
    s2();          /* S2 */
if (t2())
    x = True;      /* S3 */
else
    x = t3();      /* S4 - BUG */
```

Naish et al. Proposed Approach II

1. Proposed several metrics used in other fields and compare with O and O^p metrics
2. Observed bug localization performance is optimal using O and O^p on single bug programs
3. Empirically evaluated Siemens Test Suite, Unix and Space datasets
4. Compared on several metrics ▸ Empirical Results

Xie et al. Proposed Approach I [XiaoYuan Xie et al., 2010]

1. Proposed post-ranking of program statements evaluated with spectra metrics
2. Group statements into two suspicious groups; G_S and G_U according to heuristics
3. G_S - any statements executed by any fail test case
4. G_U - statements not executed by any fail test case

Xie et al. Proposed Approach II

1. Value of 1 is added to statement(s) in G_S
2. Small value (minimum metric value) is assigned to statement(s) in G_U
3. Found improved bug localization performance using Wong2, Wong3, Scott and M2 metrics

Debroy et al Proposed Approach I [Vidroha et al., 2010]

1. Proposed post-ranking of program statements
2. Group statements based on the fail test cases which execute respective statements (a_{ef})
3. Group statements based on same number of failed test cases (G_f) ;*f* refers to *f* fail test cases
4. Group statements executed with most fail test cases sorted at the top
5. In each group of G_f, statements sorted again based on metric value

Debroy et al [Vidroha et al., 2010] Algorithm

Input: statement-based spectra coverage, ranked statements of program with respective spectra metrics
Output: ranked statements of program that are likely to be buggy
Group statements (in descending order) based on number of failed test cases
foreach *statement s* **do**
  if *statement s is executed in f fail test cases* **then**
    Group the statement s as part of group G_f;
  end
end
Sort group G_f according to *f* fail test cases (in descending order)
Sort again the statements based on their metric value within the sorted group G_f

Debroy et al. Proposed Approach II

1. Used χsuds [Telecordia, 1998] for program instrumentation and gather coverage information
2. Evaluated on single bug programs of *Siemens Test Suite*, *grep* and *gzip* datasets
3. Using Tarantula metric [Jones et al., 2005] and Radial Basis function [Wong et al., 2008]
4. Found their strategy is more effective than without using grouping strategy

Abreu et al Proposed Approach I [Abreu et al., 2006]

1. Proposed block-based spectra metric and gathered block-based spectra coverage
2. Introduced Ochiai and Jaccard metrics and compared with Tarantula
3. Evaluated on Siemens Test Suite and observed Ochiai outperformed Tarantula metric

Gonzalez Proposed Approach [A. Gonzalez, 2007]

1. Developed Zoltar metric, based on the modification of Jaccard metric
2. Used this metric to detect dynamic invariants instead of ranking program spectra

Renieres et al. Proposed Approach I [Renieres et al., 2003]

1. Proposed to locate bugs based on *difference spectra* (differences between test cases) and *distance spectra* (using Nearest Neighbor model)
2. Implemented a tool, WHITHER in their study
3. *difference spectra* consists of two different models, namely *Intersection model* and *Union model*
4. *Intersection model*: $\cap S_p - S_f$
5. *Union model*: $S_f - \cup S_p$

Renieres et al Algorithm [Renieres et al., 2003]

Input: block-based spectra coverage of a fail test case and, all pass test cases

Output: Set of blocks of program likely to be buggy

Intersection Model -

$\cap S_p$(blocks program executed by ALL passed test cases) $- S_f$;

Union Model -

$S_f - \cup S_p$(any block program executed by the passed test cases)

Nearest Neighbor Model -

foreach *pass test case* **do**

- **Coverage Type**: Apply Hamming distance on the binary execution counts of S_p and S_f
- **Permutation Type** : Sort blocks of program in S_p and S_f based on frequency execution counts & apply Hamming distance on the test cases

end

Choose the pair of pass and fail test case with the least Hamming distance

Set of blocks of program likely to be buggy is returned

Renieres et al. Proposed Approach II

1. *Nearest Neighbor* model – one of the pass test cases that is most similar to the selected fail test cases would be chosen
2. Evaluated 109 single bug programs of Siemens Test Suite

By not examine more than 10% of the program code:

1. *Intersection model* – 1 bug
2. *Union model* – 6 bugs
3. *Nearest-neighbor using Coverage Type* – 5 bugs
4. *Nearest-neighbor using Permutation Type* – 18 bugs

Hsu et al. Proposed Approach I [Hsu et al., 2008]

1. Proposed **R**anking, **A**nalysis, and **P**attern **I**dentification for **D**ebugging (RAPID)
2. Instrumented program branch-level and gathered branch-based spectra coverage
3. Mapped each branch to respective program specta properties a_{ef}, a_{ep}, a_{nf} and a_{np}

Hsu et al. Proposed Approach II

1. Identify common branches patterns using data mining approach (BI-Directional Extension (BIDE)) [Wang et al., 2004].
2. Longest subsequence of branches is chosen and ranked most buggy
3. If the bug not found, append previous branches to next longest subsequence of branches in L
4. Performed a case study on *replace* program

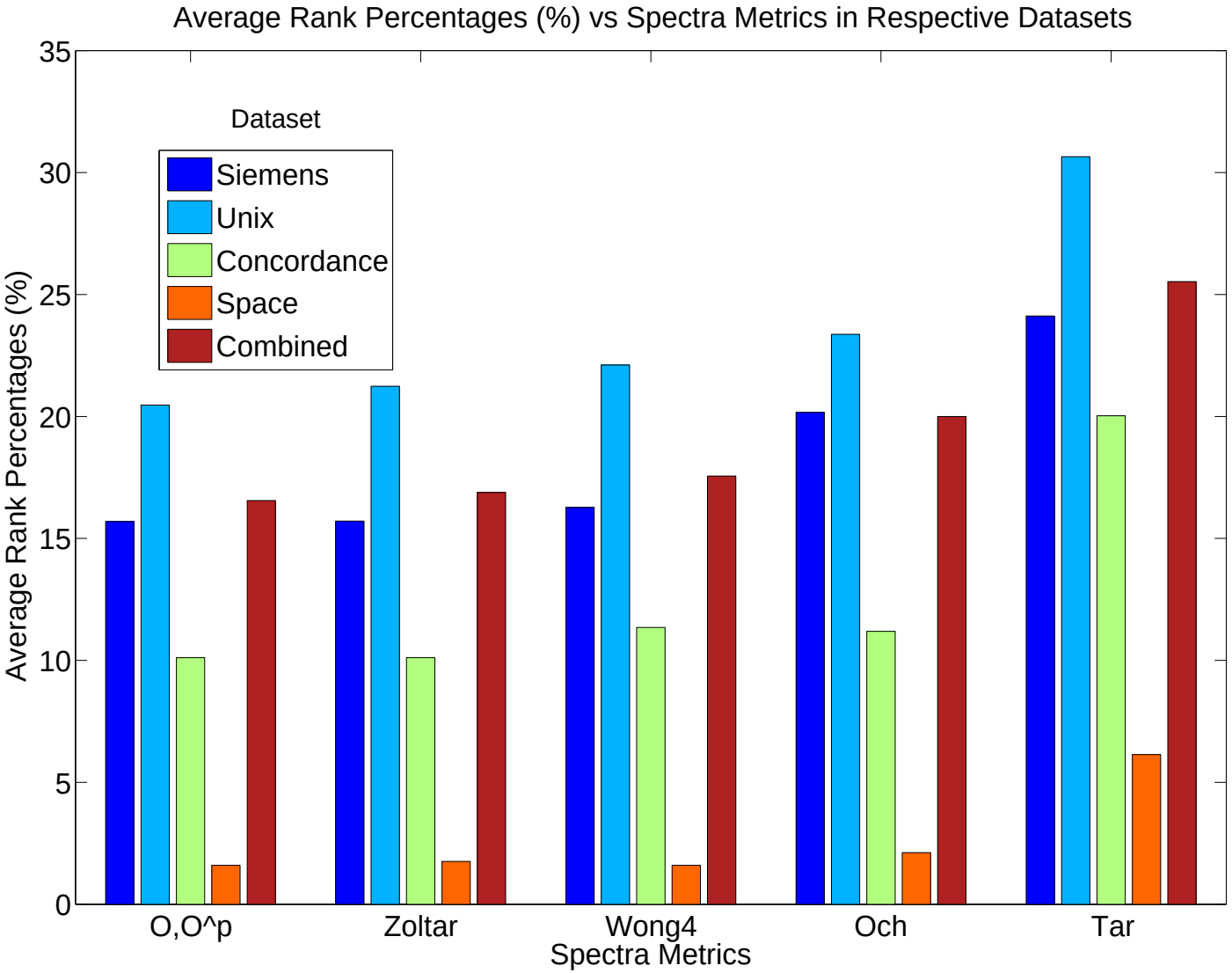
Hsu et al. Algorithm [Hsu et al., 2008]

Input: instrumented branches *B* of program evaluated with Tarantula, branch-based spectra coverage
Output: sequence of branches most likely to be buggy
Map branch *MetValue* to matching program branch in branch-based spectra coverage ;
Identify sequences of buggy branches (bug signatures);
foreach *branch-based spectra coverage of fail test case* **do**
  if *MetValue of all the mapped program branches in the test case* $<$ *0.6 (threshold value)* **then**
    Eliminate the fail test case;
  end
  else
    Store the spectra coverage of the fail test case in a new list, *L*;
  end
end
Identify common patterns of spectra coverage in *L* using BIDE;
Present longest subsequence of branches as most likely buggy;

Evaluation Results on Empirical Benchmarks of Single bug Programs

Average Rank Percentages (%) for Single bug Siemens, Unix, Concordance and Space programs

Metric	Siemens	Unix	Concordance	Space	Combined
O,O^p	15.69	20.47	10.11	1.60	16.55
Wong3	16.28	20.47	10.15	1.60	16.84
Zoltar	15.71	21.24	10.11	1.76	16.88
Wong4	16.27	22.11	11.35	1.60	17.56
Kulczynski2	16.13	22.58	10.21	2.01	17.65
JacCube	18.31	22.15	10.43	1.87	18.54
Ochiai	20.17	23.37	11.19	2.12	20.00
Jaccard	23.47	23.85	17.68	3.19	22.15
CBI Log	24.46	29.59	22.63	6.01	25.37
Tarantula	24.12	30.65	20.03	6.22	25.53
Ample	31.39	28.69	27.53	6.62	28.63
Russell	28.36	31.99	21.03	17.30	28.85
Binary	28.36	31.99	21.03	17.30	28.85

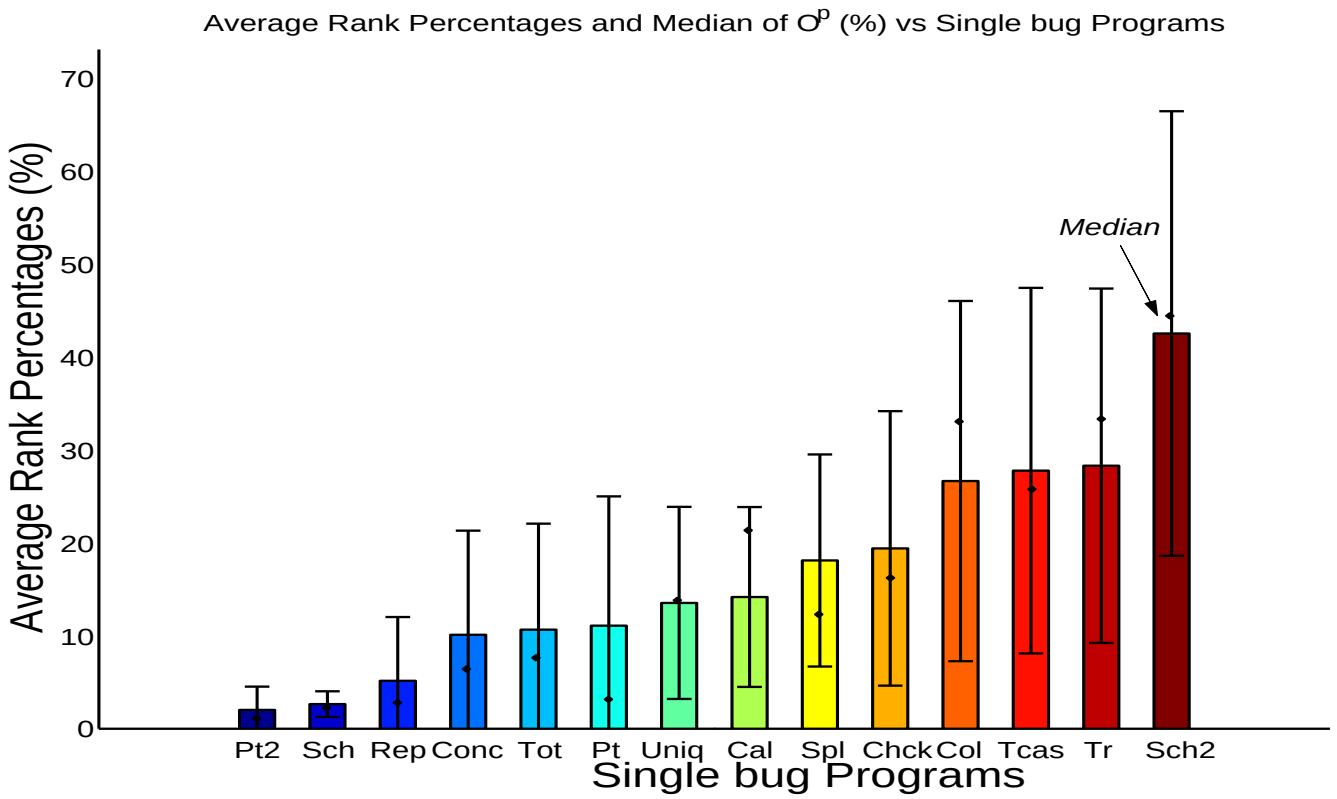

Average Rank Percentages (%) vs Spectra Metrics in Respective Datasets
Dataset
Siemens
Unix
Concordance
Space
Combined
Average Rank Percentages (%)
0
5
10
15
20
25
30
35
O,O^p
Zoltar
Wong4
Och
Tar
Spectra Metrics

Rank Percentages

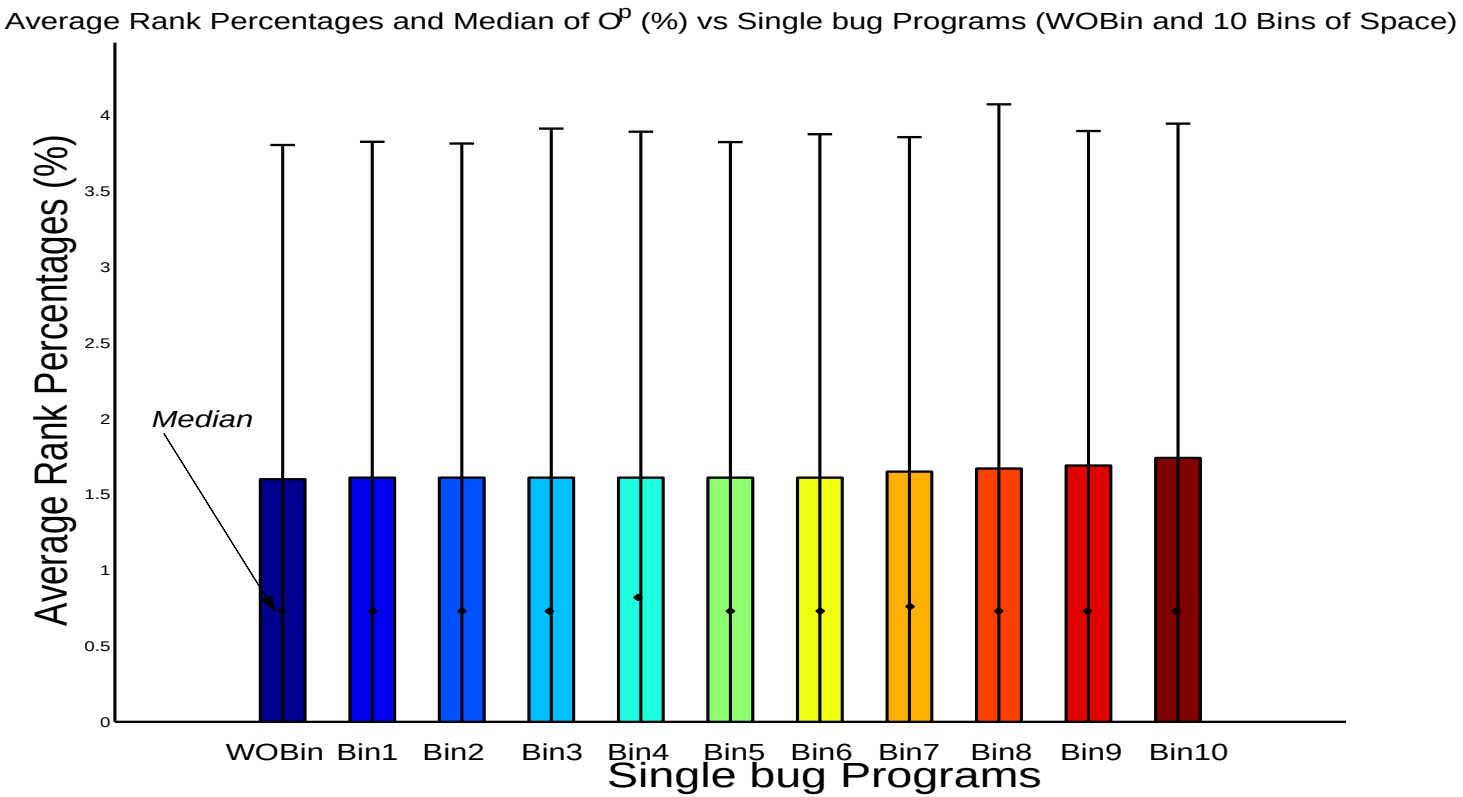
Results of Best, Average, Worst, Median, 1st Quartile and 3rd Quartile Rank Percentages (%) for Siemens Test Suite (Single bug Programs)

Metric	Best	Average	Worst	Median	1st Quartile	3rd Quartile
O, O^p	**10.69**	**15.69**	**20.68**	**5.80**	**2.34**	**24.57**
Zoltar	10.72	15.71	20.71	6.10	2.34	24.57
Kulczynski2	11.13	16.13	21.12	6.25	2.34	24.57
Wong4	11.27	16.27	21.28	6.85	2.34	25.78
Wong3	11.29	16.28	21.28	5.80	2.34	25.78
Ochiai	15.17	20.17	25.16	9.95	2.76	27.34
Jaccard	18.48	23.48	28.47	17.48	2.80	35.89
Tarantula	19.13	24.12	29.12	18.62	2.80	38.69
CBI Log	18.79	24.46	30.13	19.09	2.80	40.57
Russell	0.90	28.36	55.82	29.87	21.77	37.71
Overlap	0.90	28.39	55.88	29.87	21.77	37.71
Ample	25.82	31.39	36.96	18.30	3.04	67.97

Breakdown of Average Rank Percentages for Single bug Programs on O^p



Breakdown of Average Rank Percentages for Single bug Programs (Space) on O^p



▸ End of Empirical Results

Santelices et al. Proposed Approach I [Santelices, R. et al., 2009]

1. Investigated 3 types of spectra; statement-based spectra, branch-based spectra and DU-PAIR spectra
2. Used DUA-FORENSICS tool [Santelices, R. et al., 2007] for program instrumentation
3. Evaluated on Java programs using Ochiai metric for statement ranking
4. Use similar approach [Hsu et al., 2008] for branch and DU-PAIR instrumentation

Santelices et al. Proposed Approach II

1. DU-PAIR spectra showed most improvement in bug localization performance
2. Time incurred to instrument DU-PAIR took longer than statement-based and branch-based spectra.

On average (all Java programs), rank percentages:

1. statement-based - 11.49%
2. branch-based - 10.24%
3. DU-PAIR - 9.02%

Hao et al. Proposed Approach Algorithm [Hao, D. et al., 2008]

Input: statement-based spectra coverage
Output: $Appear_s$ and $Fail_s$
Find the membership grade of test case t ;
$\mu_t = 1/\sum_{k=1}^{N_s(P)} e_{s,t}$ // $N_s(P)$-total program statements
;
Apply conditional probability [Newmark, J., 1988];
foreach *statement s in program* **do**
    Find the set of test cases that execute statement s, $Appear_s$;
    Find the set of test cases that execute statement s and fail, $Fail_s$;
end

Hao et al. Proposed Approach Algorithm (continued)

Input: $Appear_s$ and $Fail_s$
Output: ranked program statements likely to be buggy
foreach *statement s in program* **do**
    Calculate the maximum of the membership grade for statement s **in** $Appear_s$;
    Determine the maximum membership grade of test t in $Appear_s$;
    Aggregate the maximum of the membership grade for statement s to A_s;
    Calculating the maximum of the membership grade for statement s **in** $Fail_s$;
    Determine the maximum membership grade of test t in $Fail_s$;
    Aggregate the maximum of the membership grade for statement s to F_s;
    Calculate suspiciousness of statement, s as $P(s) = |F_s|/|A_s|$;
end
Rank statements based on suspiciousness $P(s)$ in decreasing order;

Hao et al. Proposed Approach I

1. Proposed fuzzy set approach [Zadeh, Lotfi A., 1965] where membership grade is computed for each test case
2. Evaluated on Siemens Test Suite [Do et al., 2005], Tiny C Compiler (TCC) [Bellard, F., 2010], desk calculator (DC) [Morris, R. et al., 1983] and Counter of directory sizes (CNT) [CNT, 2010] datasets

Hao et al. Proposed Approach II

1. Compared with other approaches namely Dicing [Agrawal, H. et al., 1995] and Tarantula [Jones et al., 2005]
2. Fuzzy set approach had similar bug localization performance as Dicing approach
3. Only outperformed Nearest Neighbor approach

Ali et al. Proposed Approach I [Ali, S. et al., 2009]

1. Investigated the accuracy of bug localization performance on human-seeded faults
2. Claimed that manually seeded fault datasets (Siemens Test Suite) is inaccurate for bug localization study
3. For example, Siemens was manually seeded faults by the researchers [SIR, 2010]

Ali et al. Proposed Approach II

1. Proposed using Concordance (consists of 13 program versions)
2. Mutant generator tool *mutgen* [Ali, S. et al., 2009] used to generate small changes of code; treated as potential bugs
3. Compare the mutant generated program code and program code written by students

Ali et al. Proposed Approach III

1. Found similar effectiveness of bug localization performance using Tarantula metric.
2. Examine within 1% of the program code of Concordance program;
3. Concordance programs written by the students - 21% of the bugs
4. Concordance programs with the bugs generated from the mutant generator - 28% of the bugs

Hua et al. Proposed Approach I[HuaJie, L. et al., 2010]

1. Proposed frequency weighting function approach
2. Use more information of the test coverage (frequency counts)
3. Use the adapted sigmoid function M to map the non-zero frequency counts
4. If no execution (frequency count is zero), the function M returns 0.

Definition (Adapted Sigmoid Function, M)

$$M(k_{st}) = \begin{cases} \frac{1}{e^{-\alpha * k_{st}+1}} & \text{if } k_{st} > 0 \\ 0 & \text{otherwise} \end{cases}$$

where k is frequency counts of statement s of test t, α is a constant value

Hua Jie L. et al. Proposed Approach II

1. Found to improve the bug localization performance compared to using binary coverage information
2. Single bug programs, the improvement of the bug localization performance using O^p ranges in average rank percentages of 0.02% to 0.86%.
3. Multiple-bug Siemens and subset of Unix programs, the improvement of the bug localization performance using Kulczynski2 ranges in average rank percentages of 1.52% to 4.08%.

Break?!?


THINGS TO DO WHILE WAITING FOR YOUR EXPERIMENT TO FINISH
(OR SIMULATION TO RUN, OR CODE TO COMPILE, OR...)
DEVELOP YOUR LATENT PSYCHIC POWERS.
c'mon, data! support my hypothesis!
CHECK TO SEE IF ANY-ONE WROTE YOU IN THE LAST 10 SECONDS.
i have no friends.
you have no new
... you have
new mail...
no new mail.
re no new
you have no
READ A COUPLE OF RESEARCH PAPERS.
ZZZZZZ...
WATCH LIFE PASS YOU BY.
compile...
compile...
JORGE CHAM © 2004
www.phdcomics.com

Slicing

1. One of the earliest approaches in debugging [Lyle, JR, 1985, Weiser, M. et al., 1986, Agrawal, H. et al., 1990].
2. Two types; static slicing and dynamic slicing

Static Slicing

1. Parts of the program (statements or blocks of program) that affect the value of particular program variable
2. Rely on information from control flow graph (CFG).
3. Weiser proposed using program slicing in static analysis to debug program code
[Weiser, M.D., 1979, Weiser, M., 1981, Weiser, M., 1982].

Dynamic Slicing I

1. Korel et al. introduced dynamic slicing using dynamic analysis [Korel, B. et al., 1990]
2. Determine the parts of the program (a set of statements or blocks of the program) that affect the value of a particular program variable
3. Relies on the respective test cases to determine the dynamic slice

Dynamic Slicing II

1. Size of a typical dynamic slice always smaller than static slice
2. Consider only parts of program (statements/blocks) that is executed by test cases
3. Better known as an execution slice

Dicing

1. Parts of the program that appear in an execution slice but not in another execution slice [Lyle, J.R. et al., 1987]

Chen et al Proposed Approach I [Chen, TY. et al., 1993]

1. Used dynamic slice to narrow down search of the bug
2. Two possible dynamic slices, successful execution slice, $Slice_{suc}$ and fail execution slice $Slice_{fail}$
3. Used test inputs of dynamic slices to determine the successful and fail execution slice

Chen et al Proposed Approach II

1. Successful slice - output value of its slicing variables correct when applied with all associated set of test input of dynamic slice
2. Fail slice - output value of its slicing variables incorrect when applied with all associated set of test input of dynamic slice

Chen et al Proposed Approach III

1. Proposed several strategies to construct dynamic dice
   1. Remove program statements in $Slice_{fail}$ executed in the $Slice_{suc}$.
   2. Remove program statements in $Slice_{fail}$ which have been executed at least once in $Slice_{suc}$
   3. Remove program statements in $Slice_{fail}$ which have been executed in all $Slice_{suc}$
   4. Remove program statements which have been executed in ALL $Slice_{fail}$ from $Slice_{suc}$.
   5. Remove program statements which have been executed in ALL $Slice_{fail}$ and $Slice_{suc}$

Chen et al Proposed Approach IV

1. Found first strategy better than second when multiple-bug of the program are in different $Slice_{suc}$
2. Found dynamic dice is more effective to locate bugs than using static dice approach [Lyle, J.R. et al., 1987]

Agrawal et al Proposed Approach I[Agrawal, H. et al., 1995]

1. Extended study of dynamic slicing and dicing, to study bug localization performance
2. Used base-version program code as *oracle* of the program
3. *successful execution slice* - slice of each test execution where output of fault-seeded program code similar to base version program code
4. *fail execution slice* - slice of each test execution where output of fault-seeded program code differs from base version program code

Agrawal et al Algorithm [Agrawal, H. et al., 1995]

Input: successful execution slices, fail execution slices
Output: present the dices to the programmer to debug the program code
Form all the possible dices by subtracting the respective successful execution slices
Compute the number of the statement(s) in the dice, **average size**
Compute the number of occurences of the buggy statement in the dice, **good dices**
Present one of the dices randomly to the programmer to debug the program code

Agrawal et al Proposed Approach II

1. χ-slice [Telecordia, 1998] used to perform slicing and dicing of the program
2. Allows programmer visualize the execution slices and dices of the program
3. **average size** - number of statements (including the bug) for each dice.
4. **good dice** - number of occurrences that the buggy statement appear in the dices

Agrawal et al Proposed Approach III

1. Programmer presented with one of the dices randomly to debug program code
2. Found **good dice** is effective to locate bugs
3. GNU Unix Sort program used in their evaluation

Wong et al. Proposed Approach I [Wong, W.E. and Qi, Y., 2004]

1. Proposed an improved dynamic slicing approach – information of data dependencies between blocks of program
2. Agrawal et al. [Agrawal, H. et al., 1995] approach could not locate bugs if executed in both successful and fail execution slices
3. Two iterative approaches; *augmentation approach* and *refining approach*
4. Evaluated Space [Do et al., 2005] programs and performed case studies on several Space program versions

Wong et al. Proposed Approach - Augmentation Approach

1. Used if bug not found in dynamic dice
2. Iteratively include additional block(s) of program in the dynamic dice until bug is found

Wong et al. Algorithm – Augmentation Approach

Input: successful execution slice, E_s and fail execution slice, E_f
Output: code segment containing bug(s)
Define Dice D: E_f-E_s ;
Set iteration, k++ ;
Construct code segment, A^k ($D \cap E_f$ - D);
if *bug is in code segment* A^k **then**
  Stop;
end
else
  Repeat *if-block*;
end
if A^k *is similar to* A^{k-1} **then**
  No new code can be augmented;
  Reach final augmented code segment;
  No further code segment is needed to be constructed;
  Stop and return E_f to the programmer to examine the bug;
end

Wong et al. Proposed Approach - Refining Approach

1. Used to remove block(s) of the program which does not contain the bug in the dynamic dice
2. Helped programmer examine lesser program code to locate bugs
3. **D**ebugging Tool Based on **E**xecution **S**lice and **I**nterblock data **D**ependency (**DESiD**) is developed and used

Wong et al. Algorithm – Refining Approach

Input: dice D, successful and fail execution slice
Output: Code segment containing bug(s)
Randomly select k successful execution slices;
Construct new dice D^k:D-$\cup$ (k^{th} successful execution slices) ;
if *bug is found in dice* D^k **then**
    Stop;
end
else
    Set $k--$;
    if *k==0* **then**
        Stop and examine code in dice D;
    end
    else
        Unless bug is found, repeat from the beginning ;
    end
end

Break?!?


AAGH!
KHAAAAAAANNNNN!!!
YOU NAMED YOUR THESIS PROJECT "KHAN"?
IT'S EASIER TO VENT. NOW SET DEBUGGERS ON KILL.
JORGE CHAM © 2009
WWW.PHDCOMICS.COM

Statistical Approaches

1. Different statistical methods proposed to locate bugs

Liblit et al Proposed Approach I[Liblit, B. et al., 2003]

1. Proposed to remotely sample predicates from user execution of program code
2. Gathered user execution information in bug report format that feeds to a database
3. Used in software vendors: Mozilla, Microsoft, GNOME and KDE
4. If web browser crashes, a typical bug report is gathered automatically from user and fed into the database
5. Enable programmers to analyse and fix the bug

Liblit et al Proposed Approach II

1. Program code instrumented with predicates (branches, return values and scalar pairs)
2. Gather random sampling of predicates of program using geometrically distributed random numbers.
3. Each predicate consists of counter variables that determines next predicate to be sampled

Predicate Remote Sampling Algorithm

```
Input: instrumented program code (predicate-based), predicate
       countdown variable indicating next predicate to be
       sampled
Output: sampled predicate(s)
Predicate Sampling;
foreach user execution of the program code do
    if (next sampled predicate countdown > current countdown
    variable) then
        Fast Path;
        Discard the current countdown variable ;
        Get the new next sample predicate countdown ;
        Decrement the value of the current countdown variable ;
    end
    if (next sampled predicate countdown == current
    countdown variable) then
        Slow Path;
        if next sampled predicate countdown reached zero then
            Perform sampling on the predicate ;
            Reset and retrieve the next sampled predicate
            countdown;
        end
    end
end
```

Liblit et al Proposed Approach III

1. Evaluated on CCRYPT program using their proposed approach
2. Found the sampled predicates are able to represent the information to help locate bug

Predicates and Its Concepts

```
S1:     if (f == NULL){
S2:             x=0;
S3:             *f;
S4:     }else     x=1;
```

1. Excerpt to show differences of `True` and `reach`
2. If predicate on S1 is `True`, program crashes
3. Predicates on S1 and its negation (S4) is `reach` if and only if S1 is executed
4. Negation of predicate on S1 is `True` if and only if S4 is executed

Statistical Bug Isolation Algorithm [Liblit, B. et al., 2005]

Input: instrumented predicates (if-then-else, return values or scalar-pairs scheme), user executions represented as feedback report *R*

Output: Predicates ranked according to Importance(Pred) based on *F(Pred)*, *Increase (Pred)* and *CBI Log (Pred)*

Perform sparse random sampling approach on instrumented predicates [Liblit, B. et al., 2003] in *R*;

Gather the predicate-based spectra coverage information for the sampled predicates;

Compute *Failure* and *Context* for each instrumented predicate, *Pred*;

Rank the *Importance* of all the *Pred* based on *F*, *CBI Inc* and *CBI Log*

Extension of Liblit et al Proposed Approach I [Liblit, B. et al., 2005]

1. Each *Pred* has *F(Pred)*, *S(Pred)*, *F(Pred observed)* and *S(Pred observed)*
2. *F(Pred)* - number of fail tests that the *Pred* was executed and `True`
3. *S(Pred)* - number of pass tests that the *Pred* was executed and `True`

Extension of Liblit et al Proposed Approach II

1. *F(Pred observed)* - number of fail tests where the *Pred* was observed (reached)
2. *S(Pred observed)* - number of pass tests where the *Pred* was observed (reached)
3. Computed *Failure(Pred)* and *Context(Pred)* for the program
   - Predicate Metrics
4. Evaluated on MOSS, CCRYPT, BC, EXIF and RHYTHMBOX datasets [Rhythm, 2010]
5. Found their approach is able to locate bugs in these datasets, but did not compare with Tarantula metric

Spectra Metrics used in Predicate-based Spectra studies

Name	Formula
Failure(Pred)	$\frac{F(Pred)}{S(Pred)+F(Pred)}$
Context(Pred)	$\frac{F(Pred\ observed)}{S(Pred\ observed)+F(Pred\ observed)}$
CBI Inc (Pred)	Failure(Pred)-Context(Pred)
CBI Log (Pred)	$\frac{2}{\frac{1}{CBI\ \ Inc}+\frac{\log totF}{\log F(Pred)}}$
CBI Sqrt (Pred)	$\frac{2}{\frac{1}{CBI\ \ Inc}+\frac{\sqrt{totF}}{\sqrt{F(Pred)}}}$
FPC (Pred)	Failure(Pred)+Context(Pred)
OFPC	3*F(Pred)+FPC
O8FPC	10*F(Pred)+8*Failure(Pred)+Context

Liu et al. Proposed Approach I[Chao Liu et al., 2005]

1. Propose statistical model-based bug localization (better known as SOBER) to locate bugs
2. Use predicate-based spectra coverage; instrumentation is done with respect to branches and return values
3. Use information of execution counts instead of using binary information [Liblit, B. et al., 2005]
4. Capture the number of times a particular predicate, *Pred* has been executed in pass and fail test cases

Liu et al. Proposed Approach II

1. Evaluated on Siemens and BC1.06 datasets

Within 10% of program code to be examined by the programmer (130 programs in Siemens):

1. SOBER - 68 bugs
2. Liblit et al. - 52 bugs
3. Cleve et al. - 34 bugs

Liu et al. Proposed Approach III

1. SOBER uses more information of predicates than Liblit et al. [Liblit, B. et al., 2005]
2. Considered multiple evaluations of predicates (frequency counts)

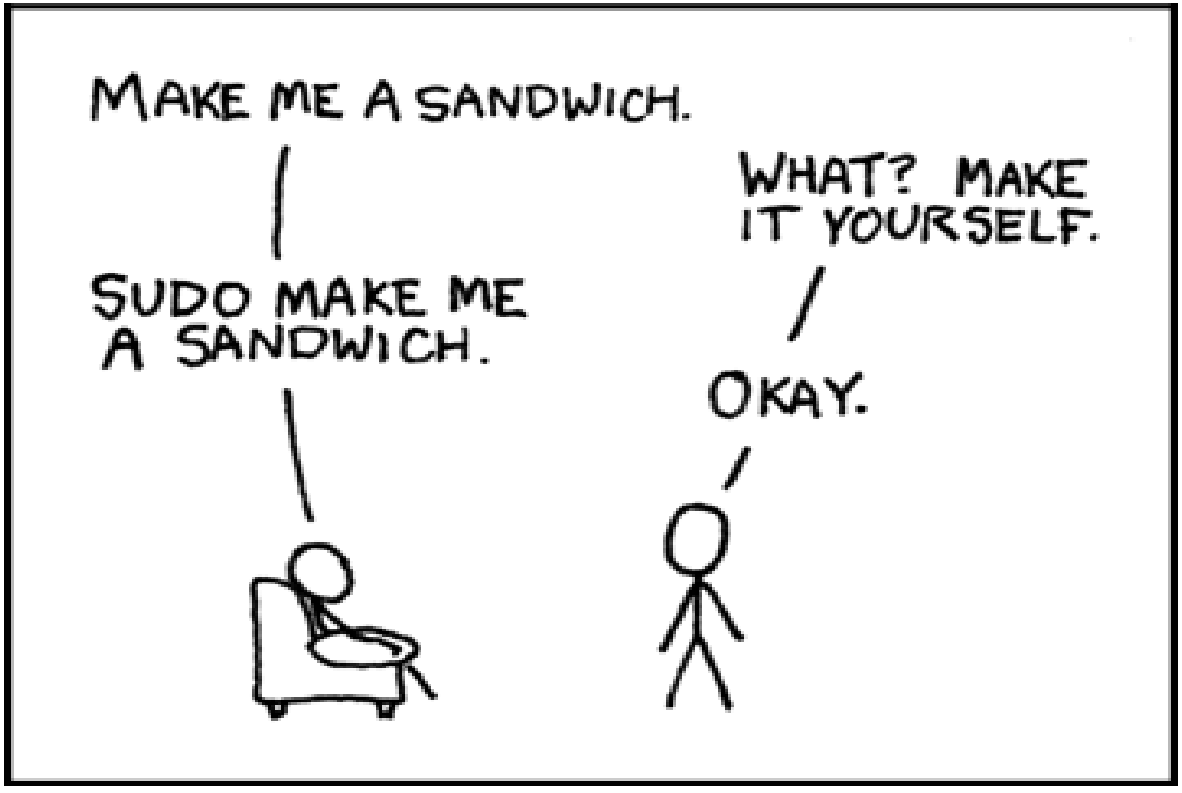
Naish et al. Proposed Approach I [Naish, Lee. et al., 2010]

1. Investigated relationship of statement-based and predicate-based spectra
2. Proposed heuristics to reconstruct predicate-based spectra (if-then-else) based on statement-based spectra
3. Evaluated on model program $ITE2_8$, Siemens Test Suite, subset of Unix [Wong, W.E. et al., 1998] and Concordance [Ali, S. et al., 2009]

Naish et al. Proposed Approach II

1. Observed improved bug localization performance for predicate-based spectra compared to statement-based spectra
2. Proposed several metrics based on Failure and Context and outperform the CBI Inc, CBI Log, and CBI Sqrt metrics [Liblit, B. et al., 2005] ▸ Predicate Metrics
3. Overall 08FPC metric outperforms other metrics in all the empirical benchmarks

Break?!?



This image is reproduced with the permission from Randall Munroe xkcd ©

Automated Debugging

1. Debugging – stepping program code (line, blocks, functions) to locate program bugs
2. State-based approaches involves automating debugging; better known as Delta Debugging [Zeller, A., 2000]
3. Isolate test case input that causes program to fail

Zeller et al. Proposed Approach [Zeller, A., 2000]

Zeller proposed to isolate the following:

1. Program input e.g input of webpage causing the web browser to fail.
2. User interaction e.g keystrokes by the user causing the program to crash.
3. Changes to the program code e.g failure inducing code changes in regression testing.

Zeller et al. Isolate Program Input Algorithm [Zeller, A., 2000]

Input: one test case that causes failure to the web browser
Output: test case input which last causes web browser failure
while *test case input can be further simplified* **do**
    Program code executed with the simplified test case input;
    if *the simplified test case input does not cause failure to the web browser* **then**
        Stop and return the previous test case input used as root cause of the web browser failure;
    end
    else
        Simplify the test case input and repeat *while-loop*
    end
end
Return the last test case input as the root cause of the failure of the web browser;

Zeller et al. Isolate Program Input Approach

1. Test case input is simplified one at a time
2. Rerun with the web browser program to check whether the web browser fails (failure still occurs)
3. Returns programmer with the last test case input that causes web browser to fail
4. Evaluated Mozilla and took 21 minutes to isolate failure in web page
5. Similar approach is used to debug failure due to user interaction

Zeller et al. Isolate Failure From Changes to Program Code

1. In regression testing, changes are made in the program code – tend to cause program code to fail
2. Known as *code differences*

Three different options :-

1. *Option 1*- Remove differences of pass test case input from fail test case input.
2. *Option 2*- Add differences of pass test case input from fail test case input to the pass test case input.
3. *Option 3*- Combination of *Option 1* and *Option 2*.

Zeller et al. Isolate Failure From Changes to Program Code (Option 1 & 2) Algorithm

Input: pass test case, fail test case, *code differences*
Output: test case input which last caused (*code differences*) to fail
while *test case input can be further simplified* **do**
    Code differences is executed with the simplified test input;
    if *the simplified test case input does not cause any failure to the program* **then**
        Stop;
        Return the previous test case input used as the root cause of program failure;
    end
    else
        Repeat *while-loop*
    end
end
Test case input could not be further simplified (input is minimum);
Return the last test case input as the root cause of program failure;
Exit;

Zeller et al. Isolate Failure From Changes to Program Code (Option 3) Algorithm

Input: pass test case, fail test case, *code differences*
Output: test case input which last caused (*code differences*) to fail
while *test case input can be further simplified* **do**
    Code differences is executed with the pass and fail test case inputs ;
    if *both pass and fail test case inputs converge* **then**
        Return the minimal difference of the last pass and fail test case input as the root cause of program failure;
        Stop;
    end
end
Return the last test case input as the root cause of program failure;
Exit;

Zeller et al. Cause-Effect Chain Approach I [Zeller, A., 2002]

1. Proposed to narrow down states (variables and values) of the program that causes program failure

Zeller et al. Cause-Effect Chain Algorithm [Zeller, A., 2002]

Input: program code that do not cause program failure P_1, program code version that causes program failure (crash) P_2, memory graph of both of these program codes
Output: cause-effect chain of failure-induced states that are relevant to the failure
Isolating relevant failure inducing state;
Use GDB [GNU GDB, 2010] to extract all the state(s) of the program code P_1 and P_2 ;
Isolate failure-induced states on both of the programs using delta debugging approach [Zeller, A., 2000];
foreach *state* **do**
    Extract the memory graphs of the state of both P_1 and P_2;
    Compare the vertices and edges from the memory graphs ;
    Variables of the state that causes failure as inputs ;
    Apply these inputs to P_1 using Delta Debugging approach [Zeller, A., 2000] ;
    Obtain last input (variables of the state) that causes failure to P_1;
end
Gather all the last input (variables of the states) as part of the cause-effect chain

Zeller et al. Cause-Effect Chain Approach II

1. Evaluated on one of the programs in GNU GCC [GCC GNU, 2010]; *fail.c* program
2. Does not require prior knowledge of the program code
3. Costly to search for the program code that causes failure

Zeller et al. Cause-Effect Chain Approach III

1. Need to navigate each state and compare with the memory graph at the same time
2. Complexity of searching the states is $2VlogSD$ where V – the number of failure inducing variables and SD – the number of state differences
3. Developed framework in a server AskIgor [AskIGor, 2010] and made public

Break?!?

Machine Learning Approaches

1. Several approaches proposed machine learning to improve existing bug localization approaches
2. Information (from test coverage) can be exploited to help locate bugs

Jones et al. Proposed Approach I [Jones, J.A. et al., 2007]

1. Proposed machine learning technique; clustering to locate bugs in multiple-bug programs
2. Cluster fail test cases that are responsible for particular bug

Two techniques to generate clusters for fail test cases:

1. Clustering based on the profiles and bug localization results, $Cluster_1$.
2. Clustering based on the bug localization results, $Cluster_2$.

Jones et al. Cluster$_1$ Algorithm

Input: a set of instrumented statement-based spectra coverage of the fail test cases
Output: cluster(s) of fail test cases responsible for the respective bug(s)
Gather sequences of branch profiles (consist of statements) executed in fail test cases in discrete-time Markov chains (DTMCs);
Apply hierarchical clustering algorithm on the DTMC for fail test cases;
foreach *level of the dendogram* **do**
  Choose the smallest absolute differences of the branch profiles in each pair of DTMC;
  Treat these fail test cases as one cluster;
  if *only one cluster formed for the level* **then**
    Stop;
  end
end

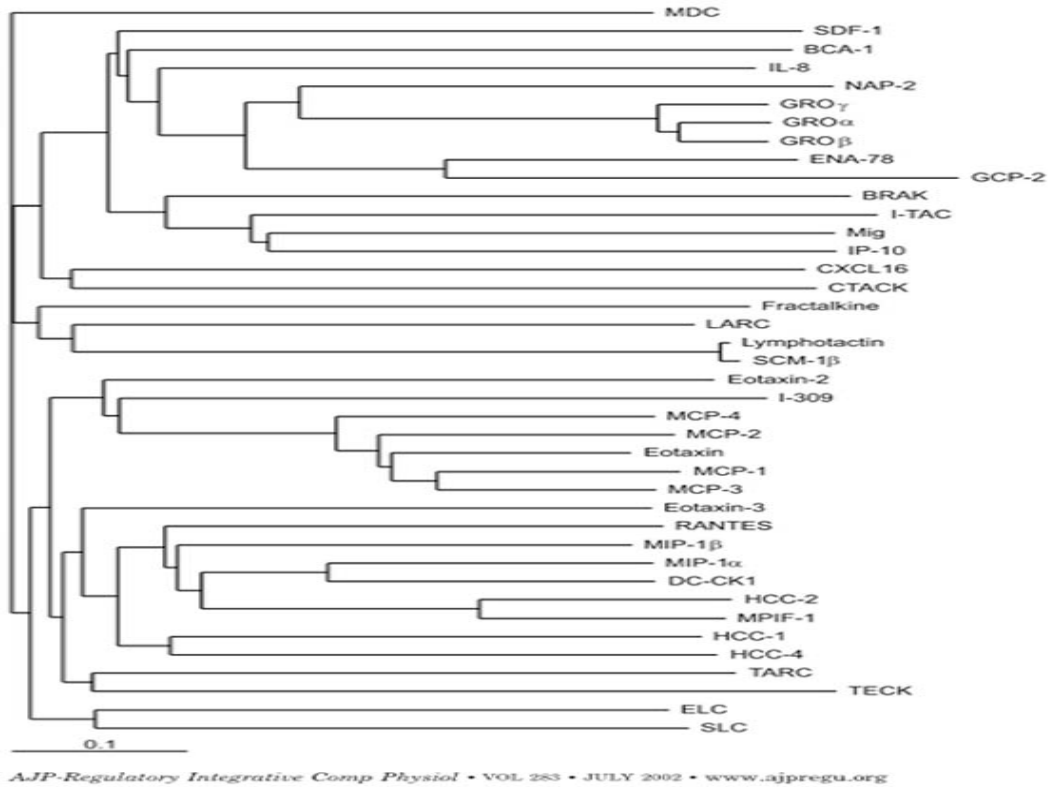
Jones et al. Proposed Approach II

1. Discrete-time Markov chains (DTMC) [Romanovskiĭ, V.I., 1970] is used; better known as *behaviour models*
2. From the algorithm, the output is dendogram [Kononenko, Igor and Matjaz Kukar, 2007]

Jones et al. Proposed Approach II

1. Once cluster(s) formed, bug localization results are used to refine clusters of fail test cases
2. Evaluate each cluster (consists of respective set of pass and fail test cases)
3. Ranking of program statements for the cluster is computed using Tarantula metric

Example of Dendogram



This image is reproduced with the permission from Virginia University [Chemokines Dendrogram, 2013]

Jones et al Cluster$_2$ Algorithm [Jones, J.A. et al., 2007]

Input: fail test cases, *specialised test suite*
Output: cluster(s) of fail test cases responsible for respective bug(s)
Evaluate and rank statements for each *specialised test suite* on Tarantula;
Perform pairwise similarity using Jaccard on the ranking statements between the specialised test suites;

1. Each *specialised test suite* contains the statement-based spectra coverage of a fail test case and all the pass test cases.
2. Evaluate proposed approaches on Space program [Do et al., 2005] using rank percentages
3. The average rank percentages evaluated using Tarantula metric on *Cluster*$_1$ - 31.50% and *Cluster*$_2$ - 26.43%

Briand et al. Proposed Approach I [Briand, L.C. et al., 2007]

1. Propose category partition method to transform test cases from the test specifications [Ostrand, T.J. and Balcer, M.J., 1988]
2. Example of transformed test case: size of an array; whether array is empty
3. Use C4.5 algorithm [Quinlan, J.R., 1993] to generate rules (forming decision trees) or better known as **RU**le-**BA**sed statement **R**anking (**RUBAR**)

Briand et al. Proposed Approach II

1. Each rule – a set of statements of program; classified as Pass/Fail
2. A rule is **Fail** - majority of Fail tests execute the set of statements of the rule
3. A rule is **Pass** - majority of Pass tests execute the set of statements of the rule
4. Positive weights - **Pass** rules; Negative weights - **Fail** rules
5. Tarantula - 15% of bugs; proposed **RUBAR** approach - 25% of the bugs in Space [Do et al., 2005]

Briand et al. [Briand, L.C. et al., 2007] Algorithm

Input: test specifications, pass and fail test cases, program code
Output: program statements with respective weights assigned
Use Category-Partition method to generate transformed test cases from the test specifications;
Classification;
Generate rules from the transformed test cases using C4.5 algorithm;
Classify rules to Pass and Fail rules;
Compute weights for Pass and Fail rules;
foreach *rule* **do**
    if *Pass rule* **then**
        Statements related to the rule assigned positive weight;
    end
    else
        Statements related to the rule assigned negative weight;
    end
end
Weights aggregated for statement and presented to the programmer;

Dickinson et al. Proposed Approach I [Dickinson, W. et al., 2001]

1. Propose clustering test cases based on similarity of test coverage using the hierarchical clustering algorithm [Murtagh, F., 1983]
2. Apply several similarity measures to form test case clusters; e.g: Euclidean distance
3. Minimal dissimilarity threshold is set to limit number of clusters

Dickinson et al. Algorithm [Dickinson, W. et al., 2001]

Input: Instrumented function caller/callee of the fault-seeded program code version, test cases
Output: clustered test cases
Execute test cases on the fault-seeded program code to gather function caller and callee of the execution coverage;
Using hierarchical clustering algorithm [Murtagh, F., 1983];
foreach *iteration or level of dendogram* **do**
    Apply similarity measures on the function caller and callee execution coverage of test cases;
    Similar execution coverage of test cases fulfills minimal dissimilarity threshold forms a cluster;
end

Dickinson et al. Proposed Approach II

1. Sampling strategies are used to find failures; *one-per-cluster sampling*, *adaptive sampling*, and *random sampling*
2. *one-per-cluster sampling* - a test case is selected for each cluster
3. *adaptive sampling* - similar to *one-per-cluster sampling*, in addition to to the selected test case is used to determine if causes program failure

Dickinson et al. Proposed Approach III

1. Percentages of failure in the clusters is evaluated where percentage of fail test cases found in each cluster
2. Evaluate on several Java programs and GNU GCC version 2.95.2 and observe bugs are not distributed in random fashion

In the smallest cluster for Java and GCC programs,

1. *adaptive sampling* – 40.26% and 5.55% of the failures
2. *one-per-cluster sampling* – 4.30% and 1.12% of the failures

Fatta el al. Proposed Approach I [Di Fatta, G. et al., 2006]

1. Proposed frequent pattern mining algorithm [Goethals,B., 2003] to locate bugs
2. Instrument program to gather function-based spectra coverage
3. Pass and fail test cases are represented in function call tree; set of functions (represented by vertices and edges) that are executed in the test cases.

Fatta et al [Di Fatta, G. et al., 2006] Algorithm

Input: function-based spectra coverage represented in function call trees
Output: functions ranked according to how likely they are buggy
Abstraction phase;
Function call trees are analysed using *zero-one-many* abstraction;
Filtering phase;
Define neighbourhood size N;
Use Frequent Pattern Mining algorithm to extract discriminative patterns according to neighbourhood size N;
Identify functions in the subtree that are executed more frequently in fail test cases than pass test cases;
Analysis phase;
Apply ranking function, f of the neighbourhood size N using

$$P(f) = \frac{\textit{support of f in all fail test cases}}{\textit{support of f in all test cases}}$$

Fatta el al. Proposed Approach II

1. *zero-one-many* abstraction proposed to reduce the number of function call trees (reduce performance and memory overhead)
2. Frequent pattern mining algorithm used to identify subtrees that are frequently executed in the pass and fail test cases – discriminative patterns

Fatta el al. Proposed Approach III

1. Evaluate on Siemens Test Suite and outperform other approaches such as the nearest neighbour [Renieres et al., 2003], cause-transition [Zeller, A., 2002], and SOBER [Chao Liu et al., 2005] approaches
2. Able to locate 22% of the buggy functions in the Siemens Test Suite with the frequent patterns of neighbourhood size 2.

Jiang et al. Proposed Approach I [Jiang, L. and Su, Z., 2005]

1. Proposed Support Vector Machine (SVM) [Steinwart, I. and Christmann, A., 2008] to locate bugs
2. Use linear and radial basis functions [Walczak, B. and et al., 1996] to determine hyperplanes of SVM
3. Combine with the cause-effect chain approach [Zeller, A., 2002] to isolate predicates of a program that cause the bug
4. CBI system [Liblit, B. et al., 2005] is used for instrumenting predicate of the program

Jiang et al. Algorithm I [Jiang, L. and Su, Z., 2005]

Input: predicate-based instrumented program, predicate-based spectra coverage of the program
Output: predicates in clusters
Classification;
Apply predicate-based spectra coverage as the input for SVM algorithm [Steinwart, I. and Christmann, A., 2008];
Assign score to the predicates of the program using random forest [Breiman, L., 2001];
Perform feature selection to choose top predicates that are most likely the bug;
Clustering;
foreach *pair of predicates chosen in feature selection* **do**
    Gather the differences of the distribution of pairs of predicates in all test execution coverage [Chao Liu et al., 2005];
    if *differences of distribution of the predicates in the test execution coverage* $< \epsilon$ **then**
        Predicates belong to the similar cluster;
    end
end

Jiang et al. Algorithm II

Input: predicates in clusters
Output: chain of predicates that are likely to be buggy
Cause-effect chain;
foreach *cluster* **do**
    if *any predicates in the cluster related to the bug* **then**
        Use Control Flow graph (CFG) to find paths that are related to the predicate;
        Form a chain of predicates;
    end
end
Present the chain of predicates for all the clusters to the programmer;

Jiang et al. Proposed Approach II

1. Siemens Test Suite - 74 bugs within examining 10% of the program code
2. Rhythmbox v0.6.4 dataset - 5 bugs after examining 1000 lines out of 56 484 lines of code
3. Causes performance overhead on larger size programs such as the Rhythmbox dataset

Proposed Approach of Zheng et al. I [Zheng, A.X. et al., 2006]

1. Proposed classification and feature selection approaches to narrow down predicates that are buggy [Zheng, A.X. et al., 2003]
2. In this study, proposed to use clustering approach – a bi-clustering scheme to cluster predicates of program responsible for a bug
3. Voting is proposed to determine particular predicate *Pred* to be the bug in each cluster using Q_{Pred} (quality of predicate)

Proposed Approach of Zheng et al. II

1. Vote of predicate, *Pred* in each cluster is aggregated across all test cases.
2. Predicate *Pred* with highest total votes in each cluster identified as one of the bugs of program
3. Evaluate proposed approach on Siemens Test Suite using PDG performance measure
   [Renieres et al., 2003, Cleve et al., 2005]
4. Examine more than 7% of the program code – found additional 70 bugs using SOBER; 65 bugs using CBI
   [Liblit, B. et al., 2003]

Zheng et al. Algorithm [Zheng, A.X. et al., 2006]

Input: predicate-based spectra coverage (frequency counts) of sampled predicates
Output: predicate with the highest votes for each cluster
// Q_{Pred} as quality of each predicate, *Pred*
Inferring Truth Probabilities;
Sampled predicates are inferred with truth probabilities using graphical model;
Generate clusters of predicates using spectral clustering algorithm [Ng, A. et al., 2001];
Collective Voting approach using bi-clustering scheme;
foreach *cluster of predicates formed* **do**
  foreach *predicate, Pred, in the cluster* **do**
    foreach *test case* **do**
      Compute the Q_{Pred};
      Contribution of predicate, *Pred* to respective test case is computed;
      Aggregate vote assigned by the test case for predicate, *Pred*;
    end
  end
  Predicate, *Pred* with the highest number of total votes is chosen as the most likely bug in the cluster;
end

Example of Practical Bug Localization Tools

1. CBI system [Liblit, B. et al., 2005]
2. HOLMES [Chilimbi et al., 2009]

Liblit et al. Proposed Public Deployment of CBI Approach [Liblit et al., 2004]

1. Distributed binaries for several large open source projects; e.g Evolution, Gaim, Gimp, Gnumeric, Nautilus and Rhythmbox
2. Gathered information of program when program crashes
3. Reports sent back to CBI server for further analysis of the root cause of the program failures
4. Observed most crashes occurred in Rhythmbox application

HOLMES system [Chilimbi et al., 2009]

1. Chilimbi et al. proposed using path-based spectra coverage and two debugging approaches
   1. Non-adaptive debugging
   2. Adaptive debugging
2. Used several real-world applications (e.g GCC, Apache and EDG compiler)
3. Used Microsoft Phoenix compiler framework for instrumentation of program branches and paths

Chilimbi et al. Proposed Approach – Non-Adaptive debugging

1. Uses similar framework as CBI system [Liblit, B. et al., 2005]
2. Gather user execution information (program paths), in the form of reports *R*, using sparse random sampling
3. Adapt the *Failure* and *Increase* metrics [Liblit, B. et al., 2005] to rank paths instead of predicates
4. Causes instrumentation overhead
5. Similar to Liblit et al. [Liblit et al.,2005] where require to maintain path counters and paths sampling

Chilimbi et al. Non-Adaptive Debugging Algorithm

Input: instrumented program code (path-based) *Path*,
path-based spectra coverage
Output: paths ranked in *Importance* of *Path* (likely to be buggy)
Failure and *Context* are computed for each instrumented path of the program code, *Path*;
Rank the *Importance* based on *Failure*, *CBI Inc*, and *CBI Log* of all the *Path*;

Chilimbi et al. Adaptive Debugging Algorithm

Input: program code, user executions information of program
Output: Paths likely to be bug returned to the programmer
Monitor each program execution for failures and gather reports;
Perform static analysis based on the reports;
Compute set of functions appear in fail stack traces ;
Instrumented program code especially on the selected set of functions;
Monitor several program execution on the set of functions for failures ;
Use static analysis component model on the buggy functions ;
Narrow down the functions using *Importance*;
while *all failures not explainable to the programmer* **do**
    Invoke static analysis component to identify other fragments of code ;
end
Bug is found and stop;

Chilimbi et al. Proposed Approach I – Adaptive debugging

1. The approach did not compromise instrumentation overhead
2. Evaluate on 6 out of 7 programs of the Siemens Test Suite [Do et al., 2005] using the Microsoft Phoenix compiler [Hall, M. et al., 2009]
3. Evaluate other programs such as *GCC*, *Translate*, and *Edg*

Chilimbi et al. Proposed Approach II – Adaptive debugging

1. Evaluate approach on different spectra such as predicate, path and branch of the programs By examining 10% of the program nodes in the Siemens Test Suite,
   1. *path-based spectra* - 24 bugs
   2. *predicate-based spectra* - 14 bugs
   3. *branch-based spectra* - 0 bug
2. Concluded path-based spectra coverage performed best in locating bugs

Recent Bug Localization Research Trend I

1. Bug localization approaches involves fiddling on metrics and no theoretical support
   [XiaoYuan Xie et al., 2010, Wong, W. Eric et al., 2012]
2. More research studies combining existing (discussed) approaches
   1. Slicing and statistical bug localization approaches
      [Lei, Yan, et al., 2012, Lei, Yan, et al., 2012]
   2. Statistical bug localization and mining source code history
      [Servant, Francisco, and James A Jones., 2012]

Recent Bug Localization Research Trend II

1. Recent study investigated the relationship of different bugs (fault classes) with bug localization effectiveness [Bandyopadhyay, Aritra, and Sudipto Ghosh, 2011]
2. Attempt to use more information from test coverage; using frequency counts [HuaJie, L. et al., 2010]
3. Give weights to test cases based on the proximity among the test cases [Bandyopadhyay, Aritra, 2011]

Take Home Message I

Using Test Coverage Information Approach

1. Test coverage information is used to help narrow down the search of bugs
2. Uses mainly binary counts of test coverage information
3. Several studies proposed different spectra metrics; shown improved bug localization performance

Take Home Message II

Using Slicing Approach

1. Different dimensions of using test coverage information
2. Study relationships between test cases (pass and fail)
3. An advantage of providing more information of bugs in the program e.g related blocks or statements of program which could be executed by a pass test case but not by a fail test case

Take Home Message III

Using Statistical Approach

1. Test coverage information is combined with statistical approaches e.g sampling predicates of program
2. Statistical methods such as random sampling, standard deviation, mean and probability model were used

Take Home Message IV

Using State-based Approach

1. Automate the traditional debugging approach to isolate the last circumstance or trails of inputs that lead to causing program failure
2. Improve efficiency of debugging large program code rather having programmer to step-in the program code

Take Home Message V

Using Machine Learning Approach

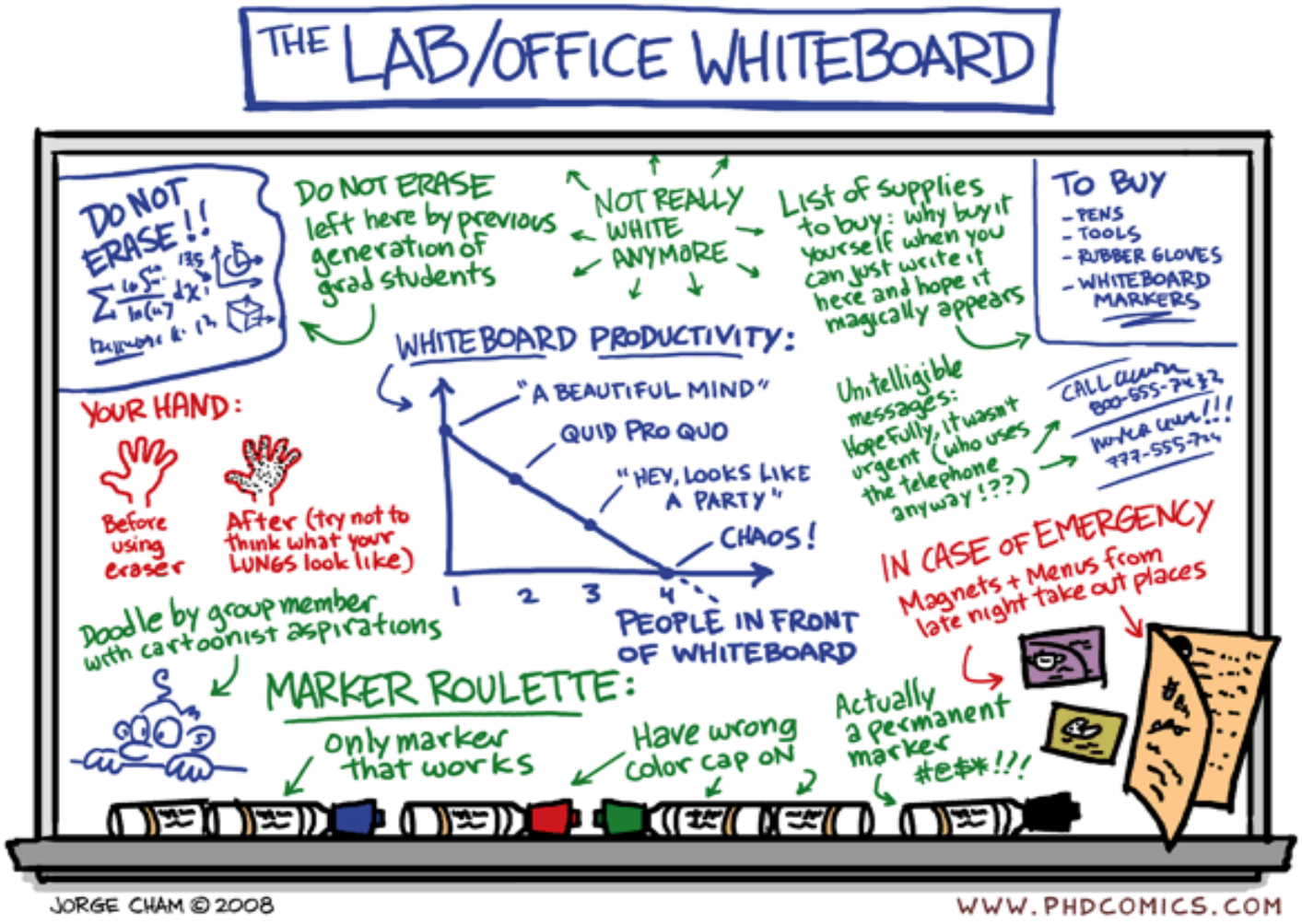
1. Exploit extra information from test coverage information
2. Learn these information and predict likelihood of bugs in the program
3. More computation required but shown improved bug localization performance overall compared to other discussed approaches

Challenges

- Explore other coverage types than statement-based spectra [Naish et al., 2009a], block-based spectra [Abreu et al., 2006], predicate-based spectra [Liblit et al.,2005] or path-based spectra [Chilimbi et al., 2009]
- Apply optimal metrics (O and O^p) and other spectra metrics into existing debugging system such as CBI [Liblit et al.,2005] and HOLMES [Chilimbi et al., 2009].
- Establish theoretical approach for optimal metrics in multiple-bug programs
- Gather more comprehensive datasets; help serve as standard de facto benchmark to compare different bug localization studies; e.g Siemens Test Suite

Question and Answer

Some Insights ...



Thank You for your attention.

Appendix of Spectra Metrics I

Name	Formula	Name	Formula
Jaccard	$\frac{a_{ef}}{totF+a_{ep}}$	Ochiai	$\frac{a_{ef}}{\sqrt{totF(a_{ef}+a_{ep})}}$
Tarantula	$\frac{\frac{a_{ef}}{totF}}{\frac{a_{ef}}{totF}+\frac{a_{ep}}{totP}}$		
Zoltar	$\frac{a_{ef}}{totF+a_{ep}+\frac{10000 a_{nf} a_{ep}}{a_{ef}}}$		
Ample	$\left\lvert \frac{a_{ef}}{totF} - \frac{a_{ep}}{totP} \right\rvert$	Ample2	$\frac{a_{ef}}{totF} - \frac{a_{ep}}{totP}$
Wong1	a_{ef}	Wong2	$a_{ef} - a_{ep}$
Wong3	$a_{ef} - h,\ \text{where } h = \begin{cases} a_{ep} & \text{if } a_{ep} \leq 2 \\ 2 + 0.1(a_{ep} - 2) & \text{if } 2 < a_{ep} \leq 10 \\ 2.8 + 0.001(a_{ep} - 10) & \text{if } a_{ep} > 10 \end{cases}$		
Wong3′	$a_{ef} - h,\ \text{where } h = \begin{cases} -1000 & \text{if } a_{ep} + a_{ef} = 0 \\ Wong3 & \text{otherwise} \end{cases}$		
CBI Inc	$\frac{a_{ef}}{a_{ef}+a_{ep}} - \frac{totF}{T}$	CBI Log	$\frac{2}{\frac{1}{CBI\ Inc}+\frac{\log totF}{\log a_{ef}}}$
CBI Sqrt	$\frac{2}{\frac{1}{CBI\ Inc}+\frac{\sqrt{totF}}{\sqrt{a_{ef}}}}$	M2	$\frac{a_{ef}}{a_{ef}+a_{np}+2(a_{nf}+a_{ep})}$

Wong4 metric

Definition (Ranking metric Wong4)

Wong4 $[(1.0) * n_{F,1} + (0.1) * n_{F,2} + (0.01) * n_{F,3}] - [(1.0) * n_{S,1} + (0.1) * n_{S,2} + 0.0001 * \frac{(a_{ef}+a_{nf})}{(a_{ep}+a_{np})} * n_{S,3}]$

$$
\begin{aligned}
\textit{where} \quad n_{F,1} &= \begin{cases} 0, & \text{for } a_{ef} = 0 \\ 1, & \text{for } a_{ef} = 1 \\ 2, & \text{for } a_{ef} \geq 2 \end{cases} \\
n_{F,2} &= \begin{cases} 0, & \text{for } a_{ef} \leq 2 \\ a_{ef} - 2, & \text{for } 3 \leq a_{ef} \leq 6 \\ 4, & \text{for } a_{ef} > 6 \end{cases} \\
n_{F,3} &= \begin{cases} 0, & \text{for } a_{ef} \leq 6 \\ a_{ef} - 6, & \text{for } a_{ef} > 6 \end{cases} \\
n_{S,1} &= \begin{cases} 0, & \text{for } n_{F,1} = 0, 1 \\ 1, & \text{for } n_{F,1} = 2 \text{ and } a_{ep} \geq 1 \end{cases} \\
n_{S,2} &= \begin{cases} 0, & \text{for } a_{ep} \leq n_{S,1} \\ a_{ep} - n_{S,1}, & \text{for } n_{S,1} < a_{ep} < n_{F,2} + n_{S,1} \\ n_{F,2}, & \text{for } a_{ep} \geq n_{F,2} + n_{S,1} \end{cases} \\
n_{S,3} &= \begin{cases} 0, & \text{for } a_{ep} < n_{S,1} + n_{S,2} \\ a_{ep} - n_{S,1} - n_{S,2}, & \text{for } a_{ep} \geq n_{S,1} + n_{S,2} \end{cases}
\end{aligned}
$$

Appendix of Spectra Metrics II

Name	Formula	Name	Formula
M1	$\frac{a_{ef}+a_{np}}{a_{nf}+a_{ep}}$	M3	$\frac{2*(a_{ef}+a_{np})}{T}$
Sørensen-Dice	$\frac{2a_{ef}}{2a_{ef}+a_{nf}+a_{ep}}$		
Kulczynski1	$\frac{a_{ef}}{a_{nf}+a_{ep}}$		
Kulczynski2	$\frac{1}{2}\left(\frac{a_{ef}}{totF}+\frac{a_{ef}}{a_{ef}+a_{ep}}\right)$		
Russell and Rao	$\frac{a_{ef}}{T}$	Hamann	$\frac{a_{ef}+a_{np}-a_{nf}-a_{ep}}{T}$
Simple Matching	$\frac{a_{ef}+a_{np}}{T}$	Lee	$a_{ef}+a_{np}$
Rogers & Tanimoto	$\frac{a_{ef}+a_{np}}{a_{ef}+a_{np}+2(a_{nf}+a_{ep})}$	Goodman	$\frac{2a_{ef}-a_{nf}-a_{ep}}{2a_{ef}+a_{nf}+a_{ep}}$
Hamming	$a_{ef}+a_{np}$	Euclid	$\sqrt{a_{ef}+a_{np}}$
Ochiai2	$\frac{a_{ef}a_{np}}{\sqrt{(a_{ef}+a_{ep})(a_{np}+a_{nf})(totF)(totP)}}$		
Ochiai3	$\frac{a_{ef}^2}{(a_{ef}+a_{nf})(a_{ef}+a_{ep})}$		
Platetsky-Shapiro	$a_{ef}+a_{ef}^2+totF-a_{ep}a_{ef}-a_{ep}a_{nf}$		
Collective Strength	$1-\frac{a_{ef}+a_{np}}{(a_{ef}+a_{ep})(totF)+(a_{nf}+a_{np})(totP)}*\frac{1-(a_{ef}+a_{ep})(totF)-(a_{nf}+a_{np})(totP)}{1-a_{ef}-a_{np}}$		
Geometric Mean	$\frac{a_{ef}a_{np}-a_{nf}a_{ep}}{\sqrt{(a_{ef}+a_{ep})(a_{np}+a_{nf})(totF)(totP)}}$		
Harmonic Mean	$\frac{(a_{ef}a_{np}-a_{nf}a_{ep})((a_{ef}+a_{ep})(a_{np}+a_{nf})+(totF)(totP))}{(a_{ef}+a_{ep})(a_{np}+a_{nf})(totF)(totP)}$		
Arithmetic Mean	$\frac{2a_{ef}a_{np}-2a_{nf}a_{ep}}{(a_{ef}+a_{ep})(a_{np}+a_{nf})+(totF)(totP)}$		

Appendix of Spectra Metrics III

Name	Formula	Name	Formula
Cohen	$\frac{2a_{ef}a_{np}-2a_{nf}a_{ep}}{(a_{ef}+a_{ep})(totP)+(totF)(a_{nf}+a_{np})}$		
Scott	$\frac{4a_{ef}a_{np}-4a_{nf}a_{ep}-(a_{nf}-a_{ep})^2}{(2a_{ef}+a_{nf}+a_{ep})(2a_{np}+a_{nf}+a_{ep})}$		
Fleiss	$\frac{4a_{ef}a_{np}-4a_{nf}a_{ep}-(a_{nf}-a_{ep})^2}{(2a_{ef}+a_{nf}+a_{ep})+(2a_{np}+a_{nf}+a_{ep})}$		
Rogot1	$\frac{1}{2}\left(\frac{a_{ef}}{2a_{ef}+a_{nf}+a_{ep}}+\frac{a_{np}}{2a_{np}+a_{nf}+a_{ep}}\right)$		
Rogot2	$\frac{1}{4}\left(\frac{a_{ef}}{a_{ef}+a_{ep}}+\frac{a_{ef}}{totF}+\frac{a_{np}}{totP}+\frac{a_{np}}{a_{np}+a_{nf}}\right)$		
Binary	0 if $a_{nf} > 0$, otherwise 1	Gower1	$\frac{a_{ef}-(a_{nf}+a_{ep})+a_{np}}{T}$
Gower2	$\frac{a_{ef}+a_{np}}{a_{ef}+a_{np}+0.5*(a_{nf}+a_{ep})}$	Gower3	$\frac{a_{ef}a_{np}-a_{nf}a_{ep}}{a_{ef}a_{np}+a_{nf}a_{ep}}$
Anderberg	$\frac{a_{ef}}{a_{ef}+2(a_{nf}+a_{ep})}$		
Added Value	$\frac{a_{ef}}{\max(a_{ef}+a_{ep},totF)}$	Interest	$\frac{a_{ef}}{(a_{ef}+a_{ep})(totF)}$
Confidence	$\max\left(\frac{a_{ef}}{a_{ef}+a_{ep}},\frac{a_{ef}}{totF}\right)$		

Appendix of Spectra Metrics IV

Name	Formula	Name	Formula
Certainty	$\max\left(\frac{a_{ef}}{a_{ef}+a_{ep}} - (a_{ef}+a_{ep}), 1-(a_{ef}+a_{ep})\right)$		
Sneath & Sokal1	$\frac{2(a_{ef}+a_{np})}{2(a_{ef}+a_{np})+a_{nf}+a_{ep}}$		
Sneath & Sokal2	$\frac{a_{ef}}{a_{ef}+2(a_{nf}+a_{ep})}$		
Phi	$\frac{a_{ef}a_{np}-a_{nf}a_{ep}}{\sqrt{(totF)(a_{ef}+a_{ep})(a_{nf}+a_{np})(totP)}}$		
Kappa	$1-\frac{a_{ef}+a_{np}-(a_{ef}+a_{ep})(totF)-(a_{nf}+a_{np})(totP)}{1-(a_{ef}+a_{ep})(totF)-(a_{nf}+a_{np})(totP)}$		
Conviction	$\max\left(\frac{(a_{ef}+a_{ep})(totP)}{a_{ep}}, \frac{(totF)(a_{nf}+a_{np})}{a_{nf}}\right)$		
Mountford	$\frac{2a_{ef}}{2(a_{ef}+a_{ep})(totF)-(2a_{ef}+a_{ep}+a_{nf})a_{ef}}$		
Klosgen	$\sqrt{a_{ef}} * \max\left(\frac{a_{ef}}{a_{ef}+a_{ep}} - totF, \frac{a_{ef}}{totF} - (a_{ep}+a_{ef})\right)$		
YuleQ	$\frac{(a_{ef}a_{np})-(a_{ep}a_{nf})}{(a_{ef}a_{np})+(a_{ep}a_{nf})}$	YuleY	$\frac{\sqrt{a_{ef}a_{np}}-\sqrt{a_{ep}a_{nf}}}{\sqrt{a_{ef}a_{np}}+\sqrt{a_{ep}a_{nf}}}$
YuleV	$\frac{(a_{ef}a_{np})-(a_{ef}+a_{ep})(totF)}{a_{ef}a_{np}+(a_{ef}+a_{ep})(totF)}$		
J-Measure	$a_{ef} * log(\frac{a_{ef}}{(a_{ef}+a_{ep})*totF}) + \max\left(a_{ep} * log(\frac{a_{ep}}{(a_{ef}+a_{ep})*totP}), a_{nf} * log(\frac{a_{nf}}{totF*(a_{nf}+a_{np})})\right)$		
Correlation	$\frac{\lvert a_{ef}a_{np}-(a_{ef}+a_{ep})(totF)\rvert-\frac{T}{2}}{\sqrt{(2a_{ef}+a_{ep})(totF+a_{np})(2a_{ef}+a_{nf})(totP+a_{ef})}}$		

Appendix of Spectra Metrics V

Name	Formula	Name	Formula
Manhattan	$1 - \frac{a_{nf}+a_{ep}}{T}$	Braun	$\frac{a_{ef}}{a_{ef}+a_{ep}}$
Baroni	$\frac{\sqrt{a_{ef}a_{np}}+a_{ef}}{\sqrt{a_{ef}a_{np}}+totF+a_{ep}}$	Coef	$\frac{a_{ef}}{a_{ef}+a_{ep}}$
Levandowsky	$1 - \frac{(a_{nf}+a_{ep})}{totF+a_{ep}}$	Watson	$1 - \frac{(a_{nf}+a_{ep})}{2a_{ef}+a_{nf}+a_{ep}}$
Jaccube	$\frac{a_{ef}}{\sqrt[3]{totF+a_{ep}}}$	NFD	$a_{ef} + a_{np}$
SokalDist	$\sqrt{\frac{a_{ef}+a_{np}}{T}}$	Overlap	$\frac{a_{ef}}{\min(a_{ef},a_{nf},a_{ep})}$
CorRatio	$\frac{a_{ef}^2}{totF(a_{ef}+a_{ep})}$	Forbes	$\frac{(T)*a_{ef}}{(a_{ef}+a_{ep})(totF)}$
Fager	$\frac{a_{ef}}{\sqrt{totF(a_{ef}+a_{ep})}} - \frac{1}{2*\sqrt{a_{ef}+a_{ep}}}$		
Simpson	$\frac{a_{ef}}{totF}$	McCon	$\frac{a_{ef}^2 - a_{nf}a_{ep}}{(totF)(a_{ef}+a_{ep})}$
AssocDice	$\frac{a_{ef}}{\min((a_{ef}+a_{ep}),totF)}$	Dice	$\frac{2a_{ef}}{totF+a_{ep}}$
Fossum	$\frac{T(a_{ef}-0.5)^2}{(totF)(a_{ef}+a_{ep})}$		
Pearson	$\frac{a_{ef}a_{np}-a_{nf}a_{ep}}{\sqrt{(totF)(a_{ef}+a_{ep})(a_{nf}+a_{np})(totP)}}$		
Dennis	$\frac{a_{ef}a_{np}-(a_{nf}a_{ep})}{\sqrt{(T)(totF)(a_{ef}+a_{ep})}}$		

End of Appendix Spectra Metrics

Surface of Respective Spectra Metrics

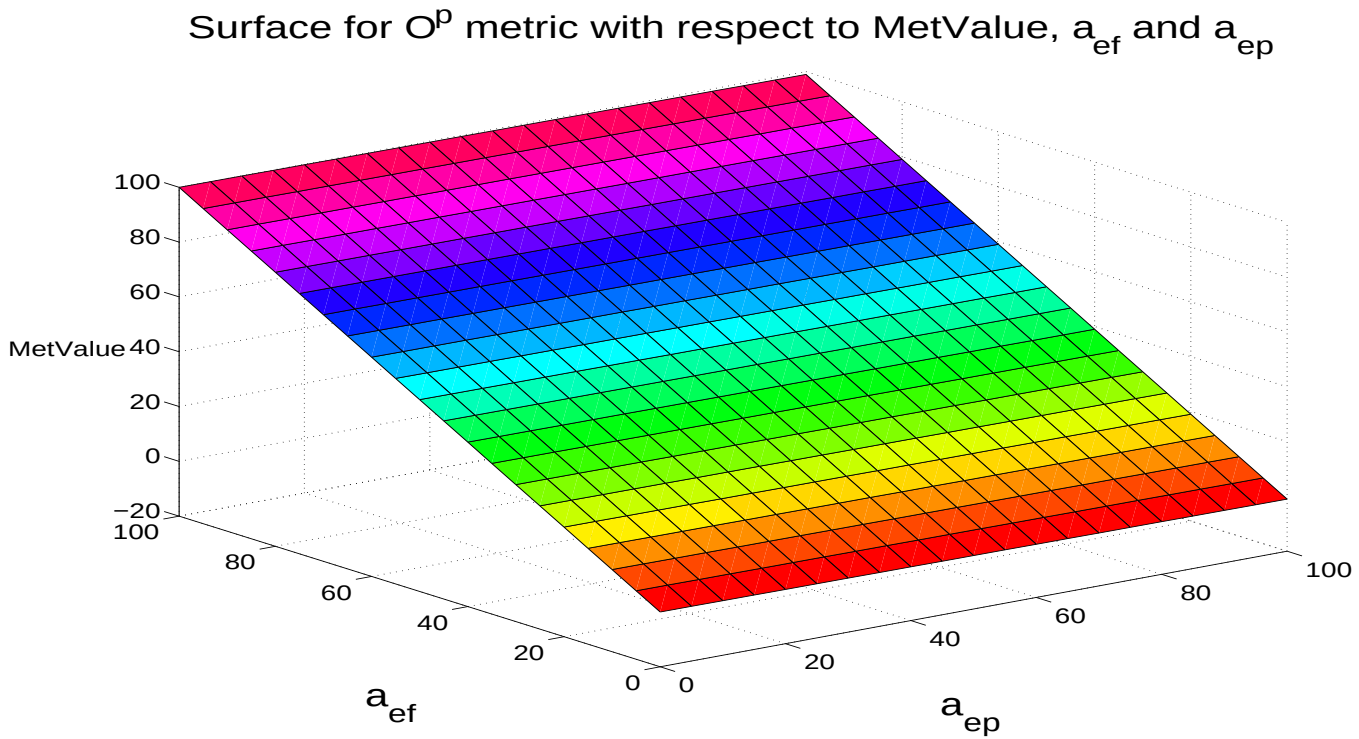


Figure: Surface for O^p metric

Surface for Rogers Metric

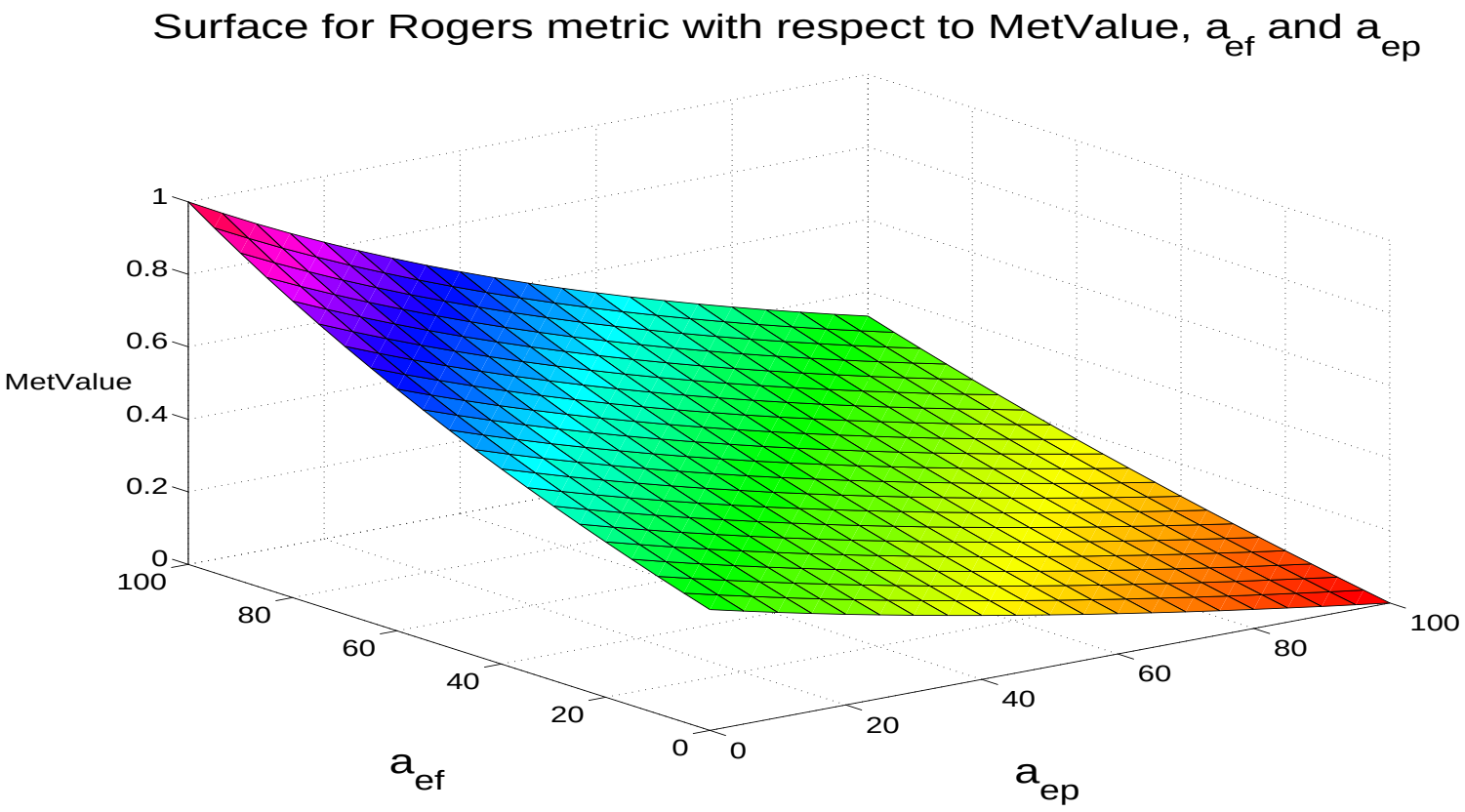


Figure: Surface for Rogers metric

Surface for Wong3 Metric

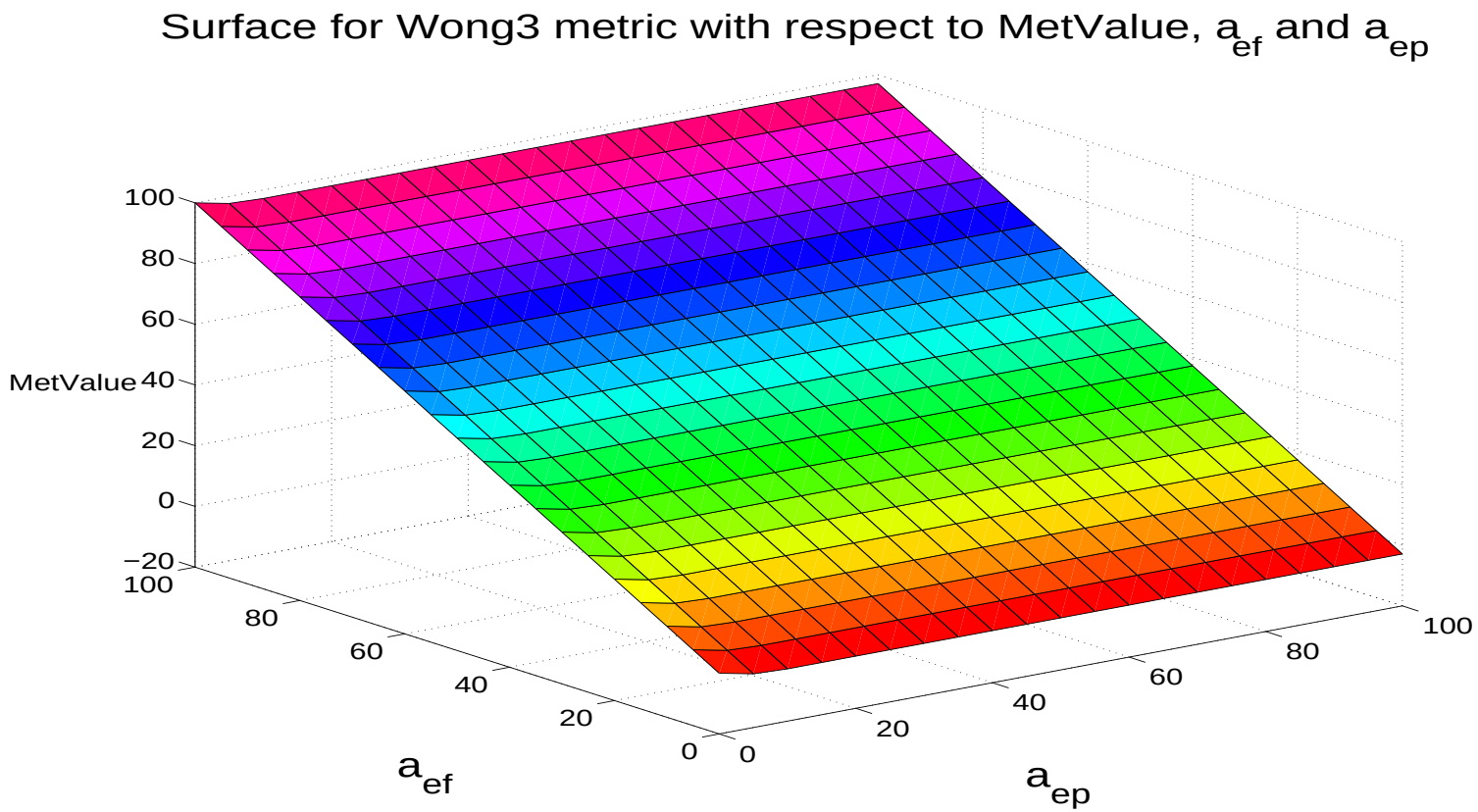


Figure: Surface for Wong3 metric

Surface for Wong4 Metric

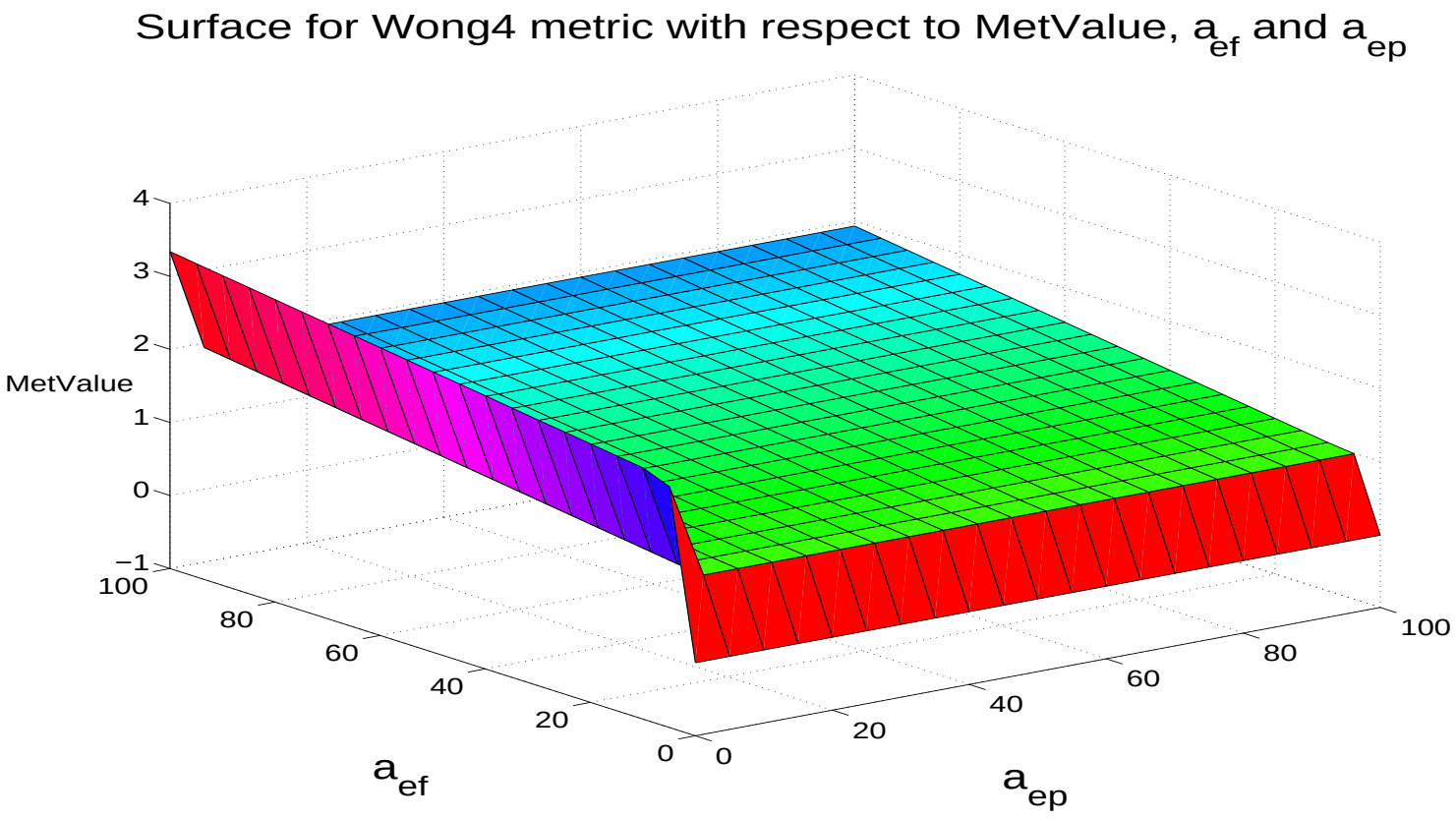


Figure: Surface for Wong4 metric

Surface for Zoltar Metric

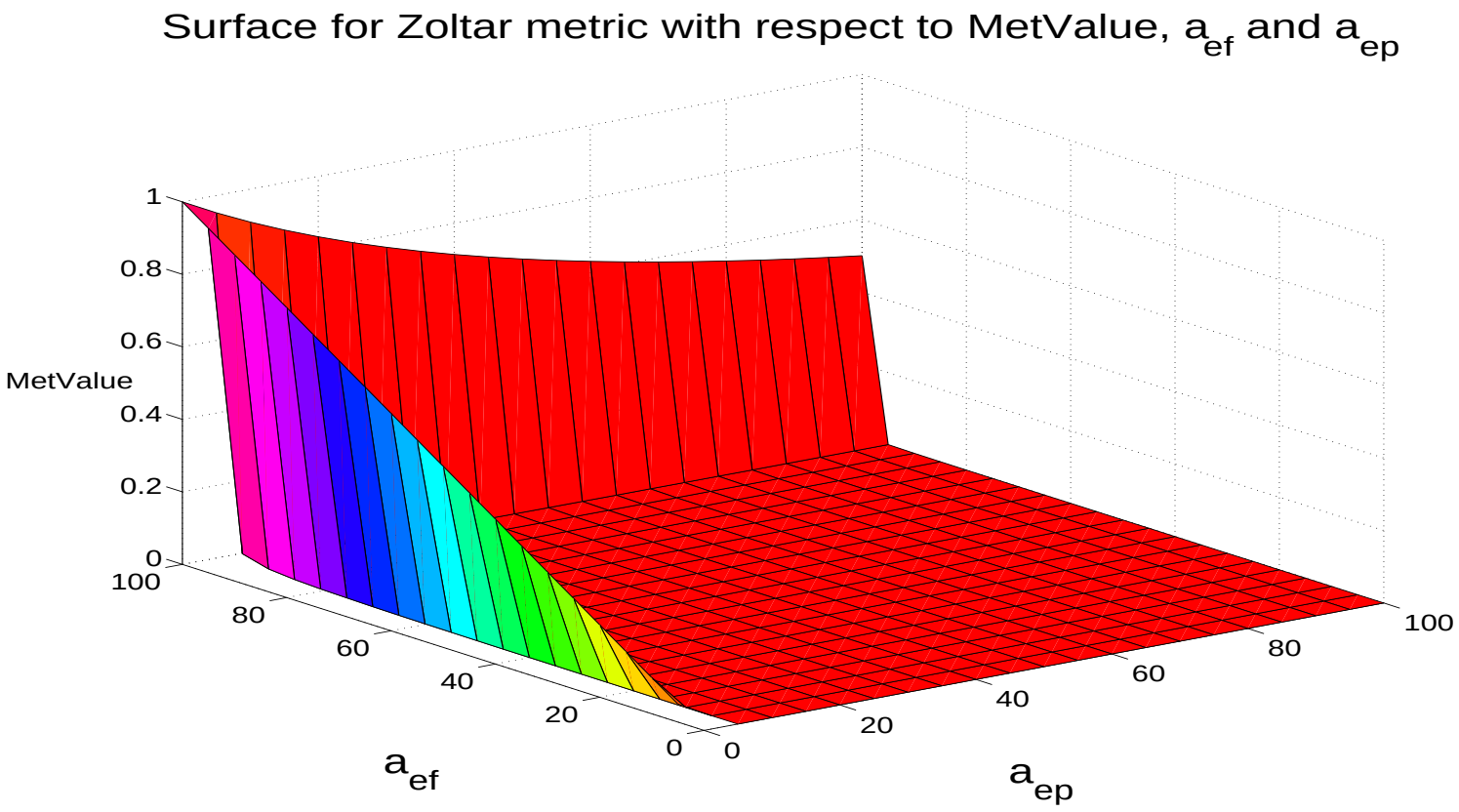


Figure: Surface for Zoltar metric

Surface for Tarantula Metric

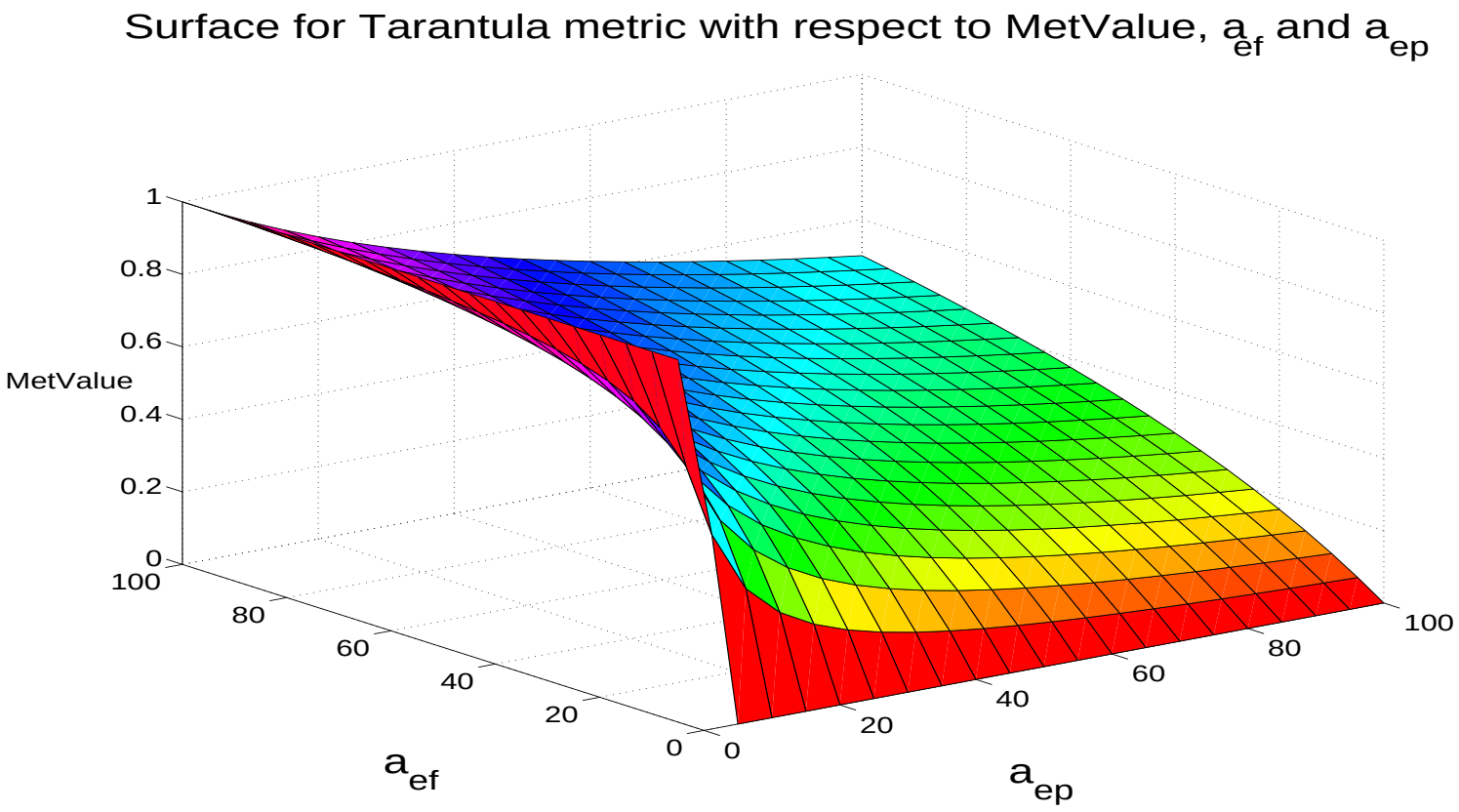


Figure: Surface for Tarantula metric

Surface for Jaccard Metric

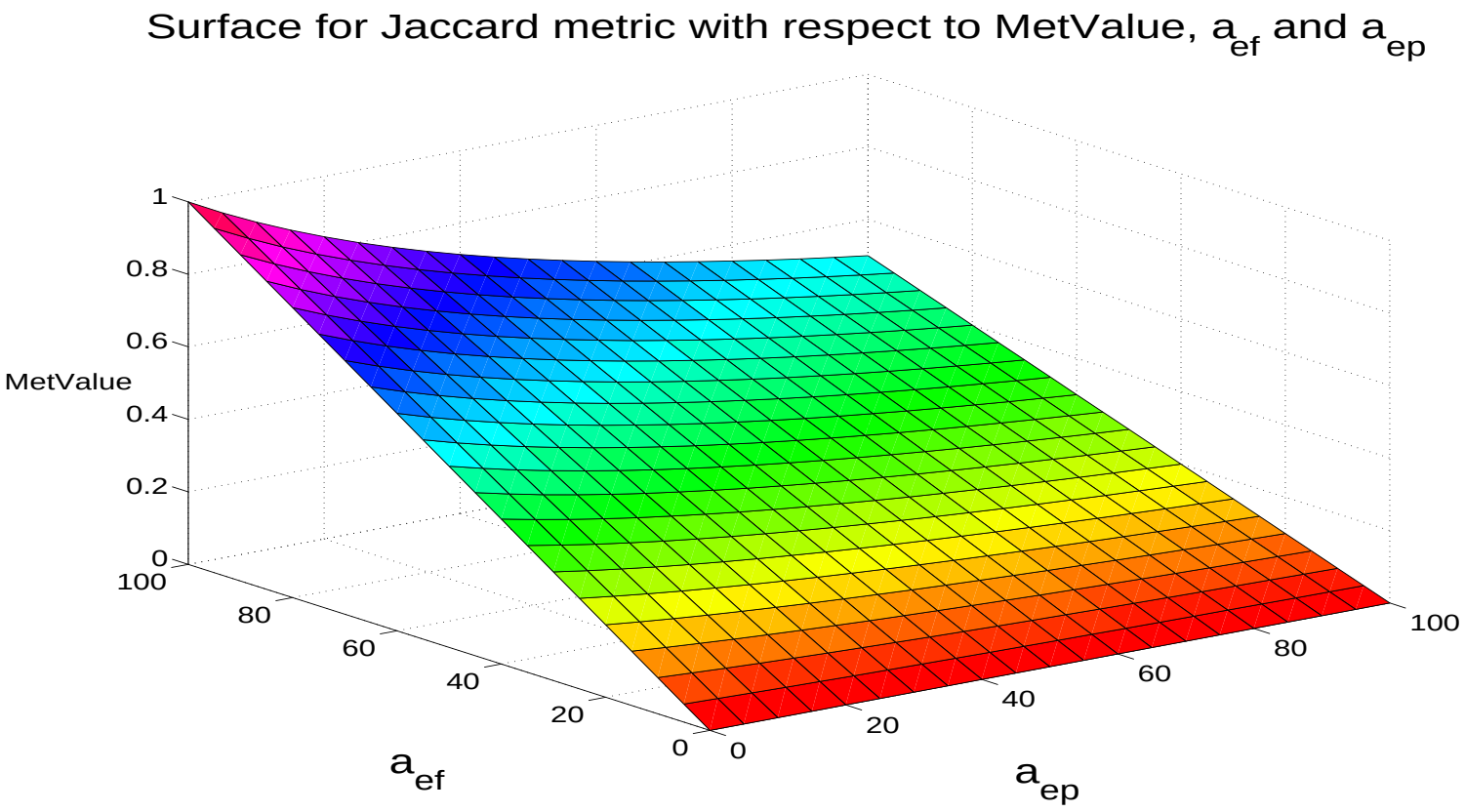


Figure: Surface for Jaccard metric

Surface for Ochiai Metric

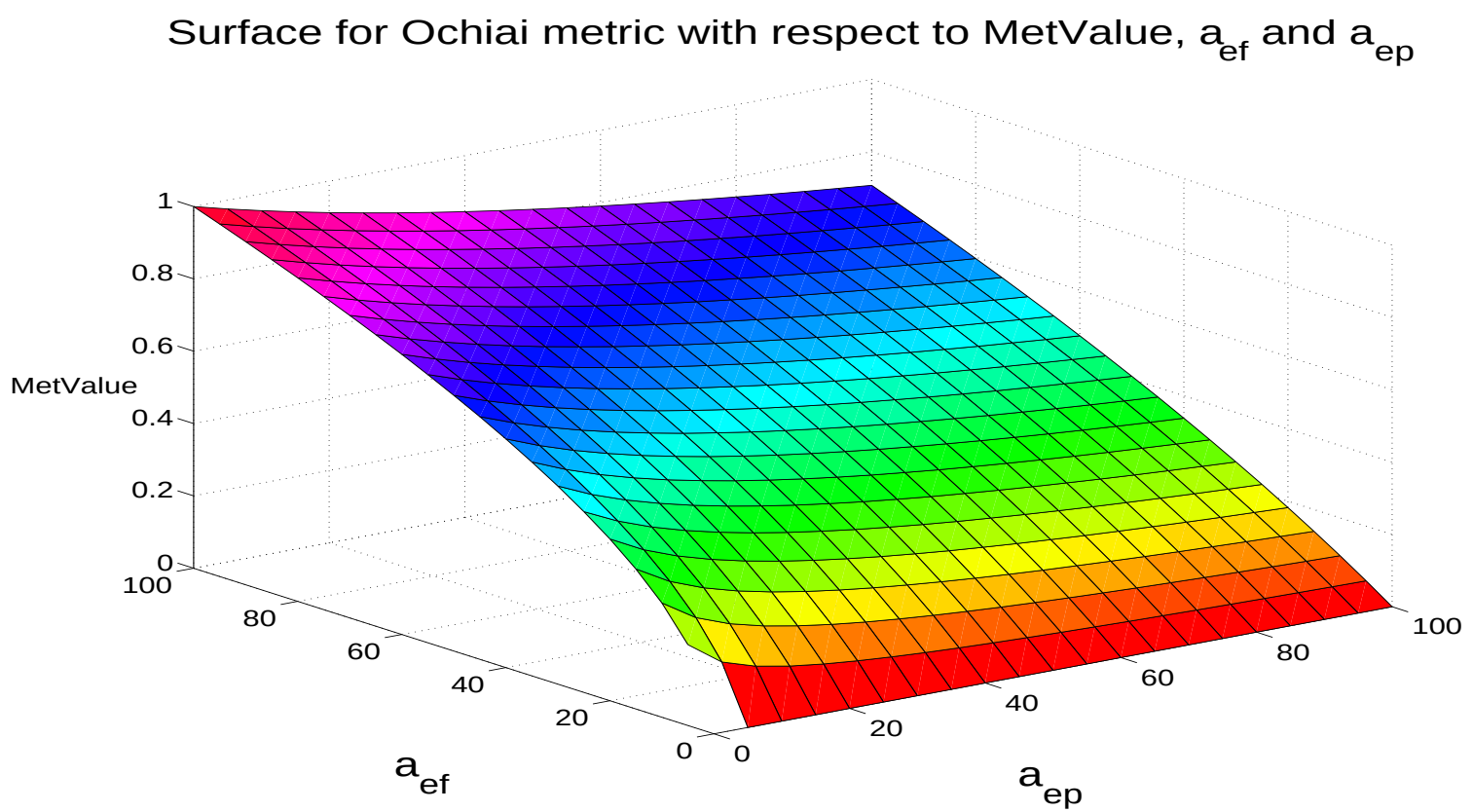


Figure: Surface for Ochiai metric

▸ End of Appendix Surface of Spectra Metrics

Evaluation Results on Multiple-bug Programs (Siemens (STS), Unix and Space test suite)

Average Rank Percentages (%) for Multiple-bug Siemens, Unix and Space programs

Metric	STSUnix(2 bug)	STSUnix(3 bug)	Space	Combined
Kulczynski2	19.53	21.94	2.61	21.44
Ochiai	20.18	22.60	2.65	22.09
Wong4	20.78	22.60	2.50	22.21
Jaccard	21.10	23.12	3.25	22.69
Zoltar	20.52	23.49	2.32	22.87
CBI Log	22.26	23.37	4.41	23.12
Ample	24.49	23.88	8.13	23.97
Wong3	22.54	25.24	2.50	24.67
O^p	22.84	25.33	2.50	24.80
Tarantula	23.13	27.23	4.51	26.39
O	24.95	29.29	2.50	28.40
Russell	32.10	28.67	17.50	29.31
Binary	34.02	32.43	17.50	32.71

Breakdown of Average Rank Percentages for Two-bug Programs

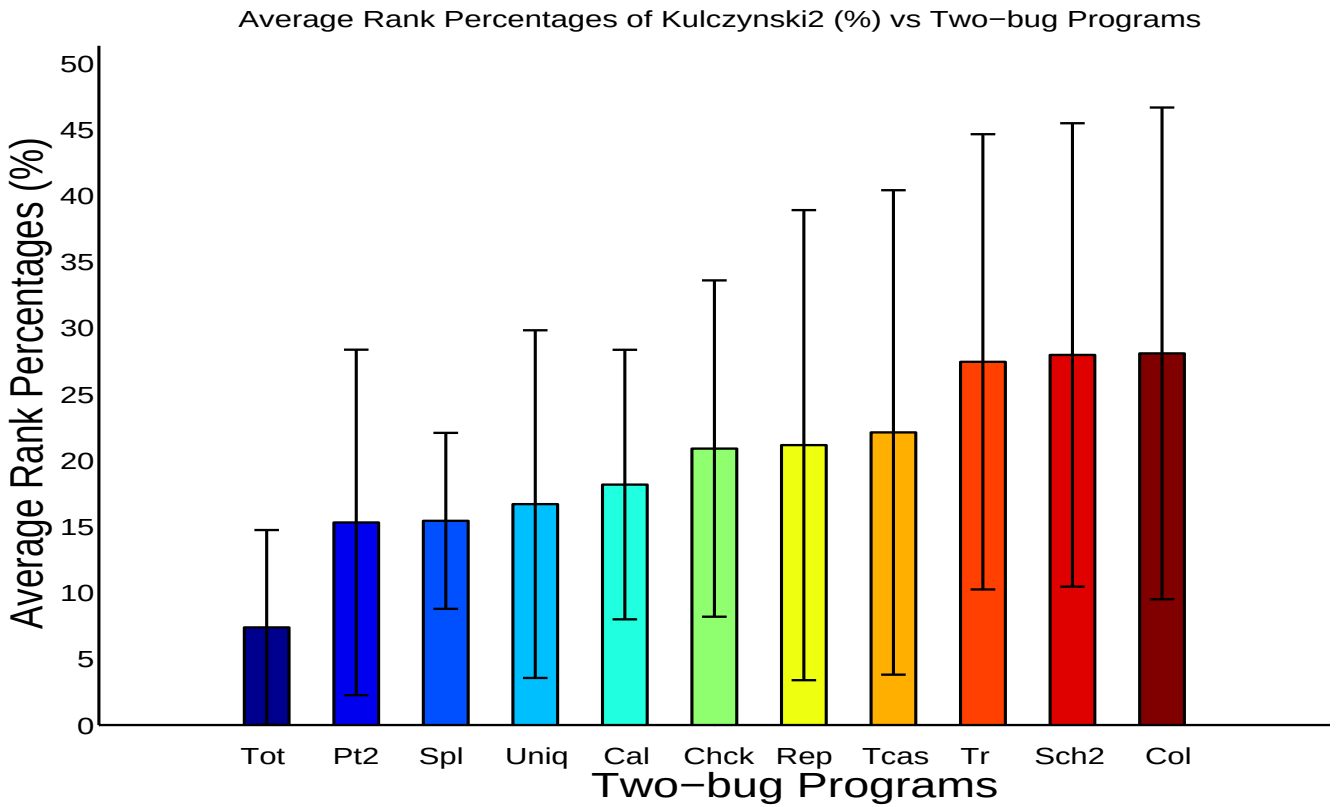


Figure: Breakdown of Average Rank Percentages for Two-bug Programs on Kulczynski2

Breakdown of Average Rank Percentages for Three-bug Programs

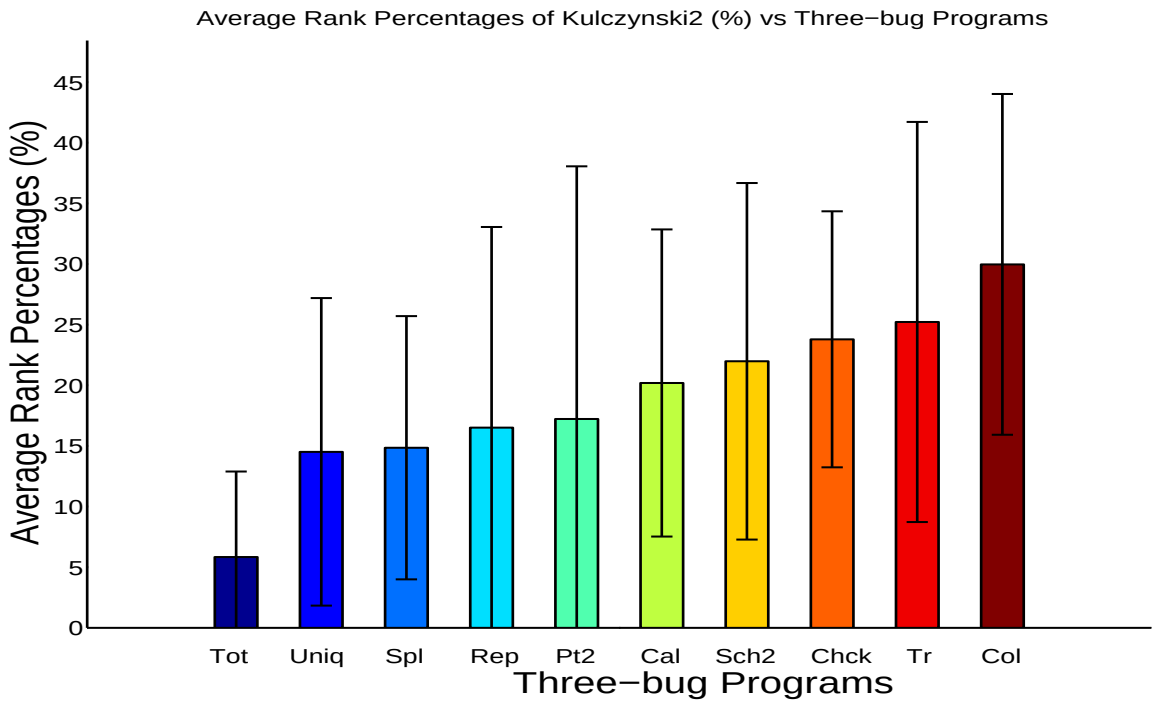


Figure: Breakdown of Average Rank Percentages for Three-bug Programs on Kulczynski2

▸ End of Appendix Evaluation of Multiple-bug Programs

Appendix of Incremental Approach on other Multiple-bug programs

Experimental Results: Results For Three-Bug Programs Siemens and Unix(Average rank percentages)

Metric	Unweighted	Weighted	10% Inc.	20% Inc.	Incre.	p-value
Kulczynski2	21.94	21.61	17.59	17.57	17.54	<0.05
Ochiai	22.60	22.25	18.22	18.20	18.19	<0.05
Wong4	22.60	21.62	18.43	18.17	17.97	<0.05
Jaccard	23.12	22.69	18.63	18.54	18.62	<0.05
Zoltar	23.49	23.44	19.32	19.40	19.35	<0.05
Ample	23.88	23.54	19.60	19.60	19.52	<0.05
Wong3	25.24	25.18	21.04	20.96	21.05	<0.05
O^p	25.33	25.25	21.05	21.02	21.08	<0.05
Tarantula	27.23	26.26	20.58	20.48	20.44	<0.05
O	29.29	28.90	22.84	22.74	22.66	<0.05

▸ End of Appendix of Incremental Approach on other Multiple-bug programs

Appendix of Incremental Approach on other Multiple-bug programs

Experimental Results: Results For Multiple-bug Space Programs (Average rank percentages)

Metric	Unweighted	Weighted	10% Inc.	20% Inc.	Incre.	p-value
Zoltar	2.42	2.42	2.04	2.05	2.05	<0.05
O	2.55	2.55	2.16	2.16	2.17	<0.05
O^p	2.55	2.55	2.17	2.18	2.18	<0.05
Wong3	2.55	2.55	2.18	2.17	2.17	<0.05
Kulczynski2	2.63	2.61	2.24	2.25	2.25	<0.05
Ochiai	2.77	2.76	2.39	2.39	2.39	<0.05
Wong4	2.80	2.75	2.36	2.38	2.37	<0.05
Jaccard	3.25	3.23	2.83	2.85	2.85	<0.05
Tarantula	4.36	4.31	3.94	3.94	3.95	<0.05
Ample	8.39	4.48	4.09	4.09	4.09	<0.05
Russell	17.85	17.85	10.52	10.45	10.50	<0.05

▸ End of Appendix of Incremental Approach on other Multiple-bug programs

Appendix of Evaluation of Frequency Weighting Function of Other Multiple-bug Programs

Table: Average Rank Percentages Result on Three-Bug Programs using Traditional (Binary) and different α values for Siemens Test Suite and Unix datasets

Metric	$\alpha =$	0.1	0.5	1	2	4	8	10	20	Bin
Kulczynski2		17.86	16.34	**16.12**	16.89	17.26	18.02	18.09	19.04	22.18

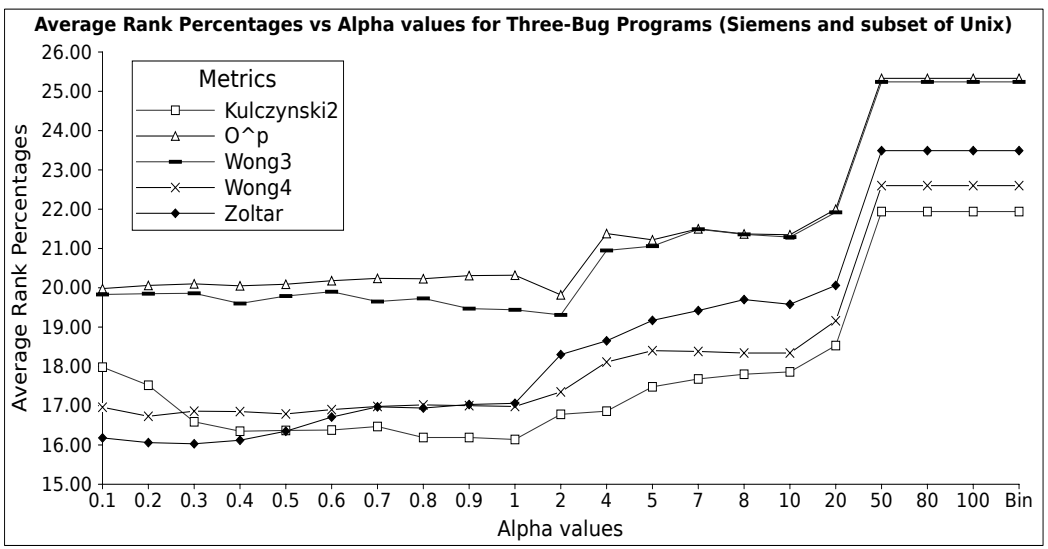


Figure: Average Rank Percentages for Different α values (Three-Bug Programs)

Appendix of Evaluation of Frequency Weighting Function of Other Multiple-bug Programs

Table: Average Rank Percentages Result on Multiple-bug Space Programs using Traditional (Binary) and different α values

Metric	$\alpha =$	0.1	0.5	1	2	4	8	10	20	Bin
Zoltar		4.01	4.46	4.28	4.15	3.33	2.70	**2.34**	2.38	2.42

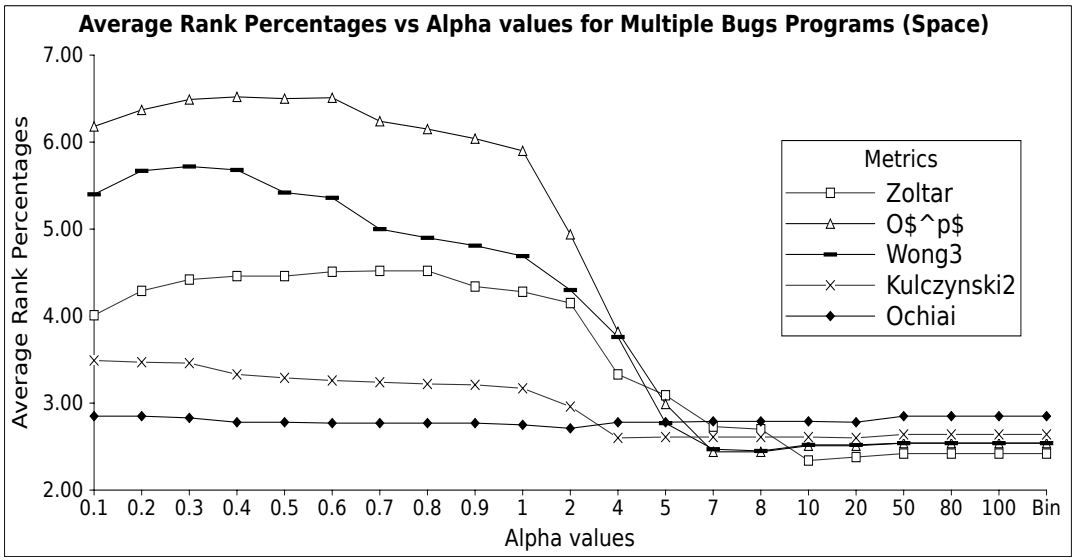


Figure: Average Rank Percentages for Different α values (Multiple-Bug Programs for Space)

References I


Hailpern, B.; Santhanam, P.
Software debugging, testing, and verification.
IBM Systems Journal, pages 4–12, 2002.


Abreu, R., Zoeteweij, P., van Gemund, A.:
An Evaluation of Similarity Coefficients for Software Fault Localization.
Proceedings of the 12th PRDC (2006) 39–46


R. Abreu, P. Zoeteweij, and A. van Gemund.
On the Accuracy of Spectrum-based Fault Localization.
pages 89–98, 2007.


W. Wong, J. Horgan, S. London, and A. Mathur.
Effect of Test Set Minimization on the Fault Detection Effectiveness of the All-Uses Criterion.

References II


M. Hollander and D. Wolfe
Nonparametric statistical methods
New York, p. 518, 1973.


Wong, W.E. and Qi, Y. and Zhao, L. and Cai, K.Y.
Effective Fault Localization using Code Coverage.
Proceedings of the 31st COMPSAC 449–456


Eric Wong, W. and Debroy, V. and Choi, B.
A family of code coverage-based heuristics for effective fault localization.
Journal of Systems and Software


L. Naish, H. Lee, and K. Ramamohanarao.
A Model for Ranking-based Software Diagnosis.
In Press ACM TOSEM, 2009.

References III


L. Naish, H. Lee, and K. Ramamohanarao.
Spectral Debugging with Weights and Incremental Ranking.

*P*roceedings of the 2009 16th APSEC, 2009. 168–175


J. Heaton
Introduction to Neural Networks for Java ,2008.


Yu, Y. and Jones, J.A. and Harrold, M.J.
An empirical study of the effects of test-suite reduction on fault localization.
*P*roceedings of the 2008 30th ICSE, 2008. 201–210


Jones, J.A. and Harrold, M.J. and Stasko, J.
Visualization of test information to assist fault localization.
Proceedings of the 24th ICSE, pages 467–477, 2002.

References IV


H. Do, S. Elbaum, and G. Rothermel.
Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact.
*E*mpirical Software Engineering, vol. 10, no. 4, pp. 405–435, 2005.


Jones, J.A. and Harrold, M.J.
Test-suite reduction and prioritization for modified condition/decision coverage.
*I*EEE Transactions on Software Engineering , 2003. 195–209

References V

Wong, W.E. and Horgan, J.R. and Mathur, A.P. and Pasquini, A.
Test set size minimization and fault detection effectiveness: A case study in a space application.
JSS , 1999. 79–89

Dan Hao, Lu Zhang,Hao Zhong,Hong Mei and Jiasu Sun.
Eliminating harmful redundancy for testing-based fault localization using test suite reduction: An experimental study.

References VI


Liblit, B., Naik, M., Zheng, A., Aiken, A., Jordan, M.:
Scalable statistical bug isolation.
Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (2005) 15–26


Chilimbi, T., Liblit, B., Mehra, K., Nori, A., and Vaswani, K. (2009).
HOLMES: Effective statistical debugging via efficient path profiling.
In *Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 34–44. IEEE Computer Society.

References VII

Neumann, P. G. (1990).
Telephone world-the crash of the at&t network in 1990.
Retrieved from
http://www.phworld.org/history/attcrash.htm.

Jones, James A, Mary Jean Harrold and John Stasko (2002).
Visualization of test information to assist fault localization..
In *Proceedings of the 24th International Conference on Software Engineering*, pages 467-477, ACM.

GIMP (2001).
Gimp
Retrieved from http://www.gimp.org.

References VIII


Renieres, M. and Reiss, SP.
Fault localization with nearest neighbor queries
In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering*, pages 30-39, IEEE.


Cleve, H. and Zeller, A.
Locating causes of program failures
In *Proceedings of the 27th International Conference on Software Engineering*, page 342-351, ACM.


Telecordia Technologies, Inc.
Telecordia software visualization and analysis toolsuite (χSuds)
In *User Manual*.

References IX


Wong, W.E. and Qi, Y. and Zhao, L. and Cai, K.Y.
Effective Fault Localization using Code Coverage
In *Proceedings of the 31st Annual International Computer Software and Applications Conference*,page 449-456, IEEE.


Hsu, H.Y. and Jones, J.A. and Orso, A.
RAPID: Identifying bug signatures to support debugging activities
In *Proceedings of the 23rd International Conference on Automated Software Enginering*, page 439-442, IEEE.

References X


Liblit, B., Naik, M., Zheng, A.X, Aiken,A., & Jordan, M.I
Public Deployment of Cooperative Bug Isolation
In *Proceedings of the 2nd International Workshop on Remote Analysis and Measurement of Software Systems* , page 57-62.


Wong,W. Eric, Vidroha Debroy, Yihao Li, and Ruizhi Gao
Software Fault Localization Using DStar (D*)
In *Proceedings of the 6th International Software Security and Reliability (SERE), 2012*, page 21-30, IEEE.

References XI

Lei, Yan, Xiaoguang Mao, Ziying Dai and Chengsong Wang

Effective Statistical Fault Localization Using Program Slices

In *Proceedings of the 36th International Computer Software and Applications Conference (COMPSAC), 2012*, page 1-10, IEEE.

Wen, Wanzhi

Software Fault Localization based on Program Slicing Spectrum

In *Proceedings of the 2012 International Computer on Software Engineering (ICSE), 2012*, page 1511-1514, IEEE.

References XII


Servant, Francisco, and James A. Jones
WhoseFault: Automatic developer-to-fault assignment through fault localization
In *Proceedings of the 2012 International Computer on Software Engineering (ICSE), 2012*, page 1511-1514, IEEE.


Bandyopadhyay, Aritra, and Sudipto Ghosh
On the Effectiveness of the Tarantula Fault Localization Technique for Different Fault Classes
In *Proceedings of the 2011 International Symposium of High-Assurance Systems Engineering (HASE), 2011*, page 317-324, IEEE.

References XIII


Bandyopadhyay, Aritra
Improving spectrum-based fault localization using proximity-based weighting of test cases
In *Proceedings of the 2011 International Automated Software Engineering (ASE), 2011*, page 660-664, IEEE.


XiaoYuan Xie and Tsong Yueh Chen and BaoWen Xu
Isolating Suspiciousness from Spectrum-Based Fault Localization Techniques
In *Proceedings of the 2010 International Conference on Quality Software (QSIC), 2010*, page 385-392, IEEE.

References XIV


Vidroha Debroy and Wong, W.E and Xiaofeng Xu and ByoungJu Choi
A Grouping-Based Strategy to Improve the Effectiveness of Fault Localization Techniques
In *Proceedings of the 10th International Conference on Quality Software (QSIC), 2010*, page 13-22, IEEE.


Jones, J.A. and Harrold, M.J.
Empirical evaluation of the tarantula automatic fault-localization technique
In *Proceedings of the 20th IEEE/ACM international Conference on Automated Software Engineering, 2005*, page 273–282, ACM.

References XV


Wong, W.E. and Shi, Y. and Qi, Y. and Golden, R.
Using an RBF neural network to Locate Program Bugs
In *Proceedings of the 19th International Symposium on Software Reliability Engineering, 2008*, page 27-36, 2008, IEEE/ACM.


A. Gonzalez
Automatic Error Detection Techniques based on Dynamic Invariants
In *Masters Thesis, Delft University of Technology, The Netherlands*, Thesis

References XVI

Wang, J and Han, J
BIDE: Efficient mining of frequent closed sequences
In *Proceedings of the 20th International Conference on Data Engineering, 2004*, page 79-90, 2004.

Santelices, R. and Jones, J.A. and Yu, Y. and Harrold, M.J.
Lightweight fault-localization using multiple coverage types
In *Proceedings of the 31st International Conference on Software Engineering, 2009*, page 56-66, 2009, IEEE

Zadeh, Lotfi A.
Fuzzy Sets
In *Information and Control Vol. 8 No. 3*, page 338-353, 1965

References XVII


Santelices, R. and Harrold, M.J.
Efficiently monitoring data-flow test coverage
In *Proceedings of the 22nd International Conference on Automated Software Engineering, 2007*, page 343-352, 2007, ACM


Hao, D. and Zhang, L. and Pan, Y. and Mei, H. and Sun, J.
On similarity-awareness in testing-based fault localization
In *Journal of Automated Software Engineering, 2008*, page 207-249, 2008, Springer


Newmark, J.
Statistics and probability in modern life, Saunders College Pub.

References XVIII


Bellard, F.
Tiny C Compiler
Retrieved *http://bellard.org/tcc/*


CNT
CNT
Retrieved *http://sourceforge.net*


Morris, R. and Cherry, L.
DC- An Interactive Desk Calculator
In *UNIX time-sharing system: UNIX programmer's manual*,1983

References XIX

Agrawal, H. and Horgan, JR and London, S. and Wong, WE and Bellcore, M.
Fault localization using execution slices and dataflow tests
In *Proceedings of 6th International Symposium on Software Reliablity Engineering, 1995*, page 143-151, 1995.

Ali, S. and Andrews, J.H. and Dhandapani, T. and Wang, W.
Evaluating the accuracy of fault localization techniques
In *2009 IEEE/ACM International Conference on Automated Software Engineering, 2009*, page 76-87, 2009, IEEE

Software-artifact Infrastructure Repository
SIR
Retrieved *http://sir.unl.edu/php/index.php*

References XX


Andrews, JH and Briand, LC and Labiche, Y.
Is mutation an appropriate tool for testing experiments?
In *Proceedings of the 27th International Conference on Software Engineering, 2005*, page 402-411, 2005, ACM


HuaJie Lee and Lee Naish and Kotagiri Ramamohanarao
Effective Software Bug Localization Using Spectral Frequency Weighting Function
In *Proceedings of the 34th Annual IEEE Computer Software and Applications Conference, 2010*, page 218-227, 2010, IEEE

References XXI

Lyle, JR
Evaluating variations on program slicing for debugging
In *Dissertation Abstracts International Part B: Science and Engineering[DISS. ABST. INT. PT. B- SCI. & ENG.], 1985*,

Weiser, M. and Lyle, J.
Experiments on slicing-based debugging aids
In *Empirical studies of programmers*, page 187–197, 1986

Agrawal, H. and Horgan, J.R.
Dynamic program slicing
In *Proceedings of the ACM SIGPLAN Notices Vol. 25 No. 6*, page 246-256, 1990, ACM

References XXII

Weiser, M.D.
Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method
In *Journal of Ann Arbor, MI*, 1979

Weiser, M..
Program slicing
In *Proceedings of the 5th International Conference on Software Engineering*, page 439-449, 1981, IEEE

Weiser, M..
Programmers use slices when debugging
In *Journal of Communications of the ACM Vol. 25 No. 7*, 1982, ACM

References XXIII

Korel, B. and Laski, J.
Dynamic slicing of computer programs
In *Journal of Systems and Software*, page 187-195, 1990

Chen, TY and Cheung, YY
Dynamic program dicing
In *Proceedings of Conference on Software Maintenance*, page 378-385, 1993

Lyle, J.R. and Weiser, M.
Automatic program bug location by program slicing
In *Proceedings of 2nd International Conference on Computers and Applications*, page 877-882, 1987

References XXIV


Agrawal, H. and Horgan, JR and London, S. and Wong, WE and Bellcore, M.
Fault localization using execution slices and dataflow tests
In *Proceedings of 6th International Conference on Software Reliability Engineering*, page 143-151, 1995


Agrawal, H. and Horgan, JR and London, S. and Wong, WE and Bellcore, M.
Fault localization using execution slices and dataflow tests
In *Proceedings of 6th International Conference on Software Reliability Engineering*, page 143-151, 1995

References XXV


Wong, W.E. and Qi, Y.
An execution slice and inter-block data dependency-based approach for fault localization
In *Proceedings of 11th Asia-Pacific Software Engineering Conference*, page 366-373, 2004, IEEE


Liblit, B. and Aiken, A. and Zheng, A.X. and Jordan, M.I.
Bug isolation via remote program sampling
In *Proceedings of ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, page 141-154, 2003, ACM

References XXVI


Liblit, B. and Naik, M. and Zheng, A.X. and Aiken, A. and Jordan, M.I.
Scalable statistical bug isolation
In *Proceedings of ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, page 15-26, 2005, ACM


Rhythm
Rhythm
Retrieved `http://rhythm.sourceforge.net`

References XXVII


Chao Liu and Xifeng Yan and Long Fei and Jiawei Han and Samuel P. Midkiff
SOBER: Statistical Model-based Bug Localization
In *SIGSOFT Softw. Eng. Notes Vol. 30 No. 5*, page 286-295, 2005, ACM


Di Fatta, G. and Leue, S. and Stegantova, E.
Discriminative pattern mining in software fault detection
In *Proceedings of the 3rd International Workshop on Software Quality Assurance*, 2006, ACM

References XXVIII


Jiang, L. and Su, Z.
Automatic Isolation of Cause-effect Chains with Machine Learning
In *Technical Report CSE-2005-32, University of California, Davis*, 2005


Steinwart, I. and Christmann, A.
Support Vector Machines, Springer Verlag


Goethals,B.
Survey on Frequent Pattern Mining, Univ. of Helsinki


Breiman, L.
Random forests Vol. 45 No. 1, page 5-32,2001, Springer Verlag

References XXIX


Zheng, A.X. and Jordan, M.I. and Liblit, B. and Naik, M. and Aiken, A.
Statistical debugging: simultaneous identification of multiple bugs
In *Proceedings of the 23rd International Conference on Machine Learning*, page 1105-1112, 2006, ACM


Ng, A. and Jordan, M. and Weiss, Y.
On spectral clustering: Analysis and an algorithm
In *Proceedings of the 2001 Conference of Advances in Neural Information Processing Systems 14*, page 849-856, 2001

References XXX

Hall, M. and Padua, D. and Pingali, K.
Compiler research: the next 50 years
In *Journal of Communications of the ACM Vol. 52 No. 2*, page 60-67, 2009, ACM

Wong, W.E. and Horgan, J.R. and London, S. and Mathur, A.P.
Effect of test set minimization on fault detection effectiveness
In *Journal of Software: Practice and Experience*, page 347-369, 1998, John Wiley & Sons

Zeller, A.
From Automated Testing to Automated Debugging
In *Journal of Uni Passau, Feb*, 2000

References XXXI


Zeller, A.
Isolating cause-effect chains from computer programs
In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, page 1-10, 2002, ACM


GNU GCC
GCC, the GNU Compiler Collection
In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, page 1-10, 2002, ACM
Retrieved `http://gcc.gnu.org/`

References XXXII

AskIgor - Automated Debugging Service
Retrieved
http://www.st.cs.uni-saarland.de/askigor/

Romanovskiĭ, V.I.
Discrete Markov Chains, Wolters-Noordhoff

Jones, J.A. and Bowring, J.F. and Harrold, M.J.
Debugging in parallel
In *Proceedings of the International Symposium on Software Testing and Analysis*, 2007,

References XXXIII


Ostrand, T.J. and Balcer, M.J.
The category-partition method for specifying and generating fuctional tests
In *Communications of the ACM Vol. 31 No. 6*, page 676-686, 1988, ACM


Quinlan, J.R.
C4. 5: programs for machine learning, Morgan Kaufmann


Kononenko, Igor and Matjaz Kukar
Machine Learning and Data Mining, Woodhead Publishing, 2007

References XXXIV


Dickinson, W. and Leon, D. and Podgurski, A.
Finding failures by cluster analysis of execution profiles
In *Proceedings of the 23rd International Conference on Software Engineering*, page 339-348, 2001, IEEE


Murtagh, F.
A survey of recent advances in hierarchical clustering algorithms
In *The Computer Journal Vol. 26 No. 4*, 1983, Br Computer Soc


GCC, GNU
GDB: The GNU Project Debugger
Retrieved *http://www.gnu.org/software/gdb/*

References XXXV


The University of Virginia, Biomedical Engineering
Chemokines Dendrogram
Retrieved *http://bme.virginia.edu/ley/chemokine_dendrogram.html*


Briand, L.C. and Labiche, Y. and Liu, X.
Using machine learning to support debugging with Tarantula
In *Proceedings of the International Symposium on Software Reliability Engineering*, 2007


Lee Naish and HuaJie Lee and Kotagiri Ramamohanarao
Statements versus predicates in spectral bug localization
In *17th Asia-Pacific Software Engineering Conference, APSEC 2010*, 2010, IEEE

References XXXVI


Walczak, B, and Massart, DL
The radial basis functions-partial least squares approach as a flexible non-linear regression technique
In *Journal of Analytica Chimica Acta Vol. 331 No. 3*, 1996, Elsevier


Zheng, A.X. and Jordan, M.I. and Liblit, B. and Aiken, A. and et al.
Statistical debugging of sampled programs
In *Journal of Advances in Neural Information Processing Systems, Vol. 16*, 2003