Statistical Debugging

Benjamin Robert Liblit. Cooperative Bug Isolation. PhD Dissertation, University of California, Berkeley, 2004.

ACM Dissertation Award (2005)

Thomas D. LaToza
17-654 Analysis of Software Artifacts

1

Despite the best QA efforts software will ship with bugs

Why would software be released with bugs?

2

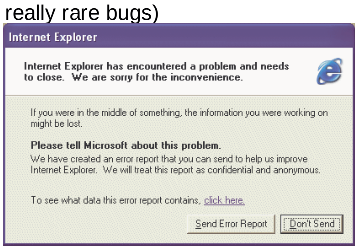
Despite the best QA efforts software will ship with bugs

Why would software be released with bugs?
- Value in getting user feedback early (betas)
- Value in releasing ahead of competitors
- Value in releasing to meet a planned launch date
- Bug doesn't hurt the user all that much

Even with much better analysis, will likely be attributes or problems hard to assure for some time

=> Free(1) testing by users!
- With real test cases (not the ones developers thought users would experience)
- By many users (might even find really rare bugs)

Result:
Send Error Report Dialog



(1) For company writing software, not users….

4

Bugs produced by error reporting tools must be bucketed and prioritized

Company (e.g. Microsoft) buckets traces into distinct bugs
- Automated tool takes stack trace and assigns trace to bug bucket
- Bug buckets: count of number of traces, stack trace for each

All bugs are not equal – can make tradeoffs
- Automated test coverage assumes all bugs are equal

- Bug that corrupts Word docs, resulting in unrecoverable work, for 10% of users
- Unlikely bug that causes application to produce wrong number in Excel spreadsheet
- Limited time to fix bugs – which should you fix?

- Frequency of bug (how many users? How frequently per user?)
- Importance of bug (what bad thing happened?)

5

But there are problems with the standard bug submission process

User hits bug and program crashes
Program (e.g. Microsoft Watson) logs stack trace
Stack trace sent to developers
Tool classifies trace into bug buckets

Problems

- WAY too many bug reports => way too many open bugs
  => can't spend a lot of time examining all of them
  - Mozilla has 35,622 open bugs plus 81,168 duplicates (in 2004)
- Stack trace not good bug predictor for some systems (e.g. event based systems)
  ⇒ bugs may be in multiple buckets or multiple bugs in single bucket
- Stack trace may not have enough information to debug
  => hard to find the problem to fix

6

What's wrong with debugging from a stack trace?

```
main()
   exif_data_save_data()
      exif_data_save_data_content()
         exif_data_save_data_content()
            exif_data_save_data_entry()
               exif_mnote_data_save()
                  exif_mnote_data_canon_save()
                     memcpy()
```

CRASH HERE SOMETIMES

```
// snippet of exif_mnote_data_canon_save()
for (i = 0; i < n->count; i++) {
   ...
   memcpy(*buf + doff, n->entries[i].data, s);
   ...
}
```

CRASH HERE SOMETIMES

- Scenario A – Bug assigned to bucket using stack trace
  - What happens when other bugs produce crash with this trace?
- Scenario B – Debugging
  - Seems to be a problem allocating memory
  - Where is it allocated?
  - Not in any of the functions in the stack trace….
  - Arg…… It's going to be a long day…..

7

Statistical debugging solves the problem - find predicates that predict bug!

Extra methods!

```
main()
  exif_loader_get_data()
    exif_data_load_data()
      exif_mnote_data_canon_load()
  exif_data_save_data()
    exif_data_save_data_content()
      exif_data_save_data_content()
        exif_data_save_data_entry()
          exif_mnote_data_save()
            exif_mnote_data_canon_save()
              memcpy()
```

(o + s > buf_size) strong predictor

CRASH HERE SOMETIMES

```
// snippet of exif_mnote_data_canon_load()
for (i = 0; i < c; i++) {
   n->count = i + 1;
   ...
   if (o + s > buf_size) return;
   ...
   n->entries[i].data = malloc(s);
   ...
}
```

(o + s > buf_size) strong predictor

8

The goal of statistical debugging

Given set of program runs

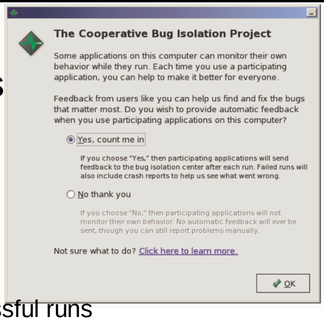
- Each run contains counters of predicates sampled at program points

Find

1. Distinct bugs in code – distinct problems occurring in program runs
2. For each bug, predicate that best predicts the bug

9

Statistical bugging technique sends reports for failing and successful runs



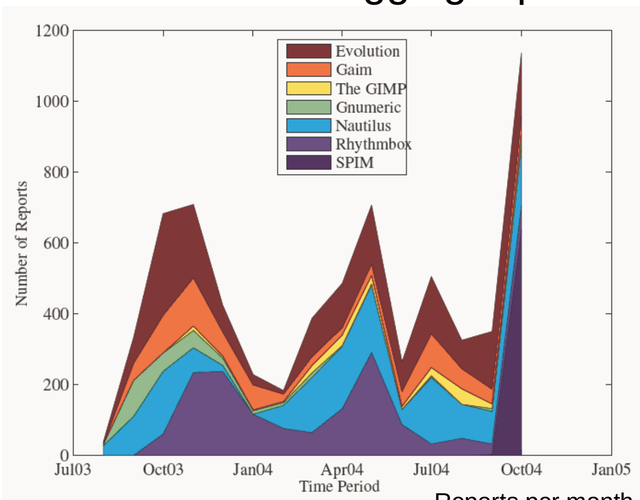
- Program runs on user computer
  - Crashes or exhibits bug (failure)
  - Exits without exhibiting bug (success)
- Counters count # times predicates hit
  - Counters sent back to developer for failing and successful runs
- Statistical debugging finds predicates that predict bugs
  - 100,000s to millions of predicates for small applications
  - Finds the best bug predicting predicates amongst these
- Problems to solve
  - Reports shouldn’t overuse network bandwidth (esp ~2003)
  - Logging shouldn’t kill performance
  - Interesting predicates need to be logged (fair sampling)
  - Find good bug predictors from runs
  - Handle multiple bugs in failure runs

10

Deployment and Sampling

11

OSS users downloaded binaries submitting statistical debugging reports



Reports per month

Small user base ~ 100??
And only for small applications
Got press on CNet, Slashdot in Aug 2003

12

Data collected in predicate counters

Fundamental predicates sampled on user computer

"$x < y$ on line 319 of `utils.c`" was observed to be true 25 times

"$x = y$ on line 319 of `utils.c`" was observed to be true 3 times, and

"$x > y$ on line 319 of `utils.c`" was observed to be true 1 time.

Infer predicates on developer's computer from fundamental predicates

"$x \geq y$ on line 319 of `utils.c`" would have been observed to be true $3 + 1$ times,

"$x \neq y$ on line 319 of `utils.c`" would have been observed to be true $25 + 1$ times, and

"$x \leq y$ on line 319 of `utils.c`" would have been observed to be true $25 + 3$ times.

13

Predicates sampled at distinguished instrumentation site program points

Branches

- if (condition) while(condition) for(; condition ;)
- Predicates – condition, !condition

Function entry

- Predicate - count of function entries

Returns

- Predicates – retVal < 0, retVal = 0, retVal > 0

Scalar pairs – assignment

- x = y
- Predicates x > z, x < z, x = z for all local / global variables z in scope

14

Sampling techniques can be evaluated by several criteria

Minimize runtime overhead for user

- Execution time
- Memory footprint

Sample all predicates enough to find bugs

- Maximize number of distinct predicates sampled
- Maximize number of times predicate sampled
- Make sample statistically fair – chance of sampling each instrumentation site each time encountered is the same

15

What's wrong with conventional sampling?

Approach 1: Every n executions of a statement
Approach 2: Sample every n statements

```
{
if (counter == 100) { check(p != NULL); counter++}
p = p->next

if (counter == 100) { check(i < max); counter++}
total += sizes[i]
}
```

Approach 3: Toss a coin with probability of heads 1/100 (“Bernoulli trial”)

```
{
if (rnd(100) == 0) { check(p != NULL); counter++}
p = p->next

if (rnd(100) == 0) { check(i < max); counter++}
total += sizes[i]
}
```

17

Instead of testing whether to sample at every instrumentation site, keep countdown timer till next sample

Consider execution trace – at each instrumentation site
- If 0, came up tails and don't sample
- If 1, came up heads and sample predicates at instrumentation site

Let the probability of heads (sampling) be p=1/5

p=1/5 of sampling at each site

Example execution trace

$$\langle \underbrace{0,0,0,0,0,1}_{6}, \underbrace{0,0,0,1}_{4}, \underbrace{0,1}_{2}, \underbrace{0,0,1}_{3}, \ldots \rangle$$

Time till next sample

Idea – keep countdown timer till next sample instead of generating each time

How to generate number to countdown from
to sample with probability p = 1/5 at every instrumentation site?

18

Instead of testing whether to sample at every instrumentation site, keep countdown timer till next sample

Consider execution trace that hits list of instrumentation sites
- If 0, came up tails and don't sample
- If 1, came up heads and sample predicates at instrumentation site

Let the probability of heads (sampling) be p=1/5

Example execution trace

time t time t+k

$$\langle \underbrace{0,0,0,0,0,1}_{6}, \underbrace{0,0,0,1}_{4}, \underbrace{0,1}_{2}, \underbrace{0,0,1}_{3}, \ldots \rangle$$

Time till next sample

What's the probability that the next sample is at time t+k?

Time t: (1/5)
Time t+1 (4/5) * (1/5)
Time t+2 (4/5)^2 * (1/5)
Time t+3 (4/5)^3 * (1/5)
Time t+k (4/5)^k * (1/5)

=> p * (1 – p)^k => Geometric distribution
Expected arrival time of a Bernoulli trial

19

Generate a geometrically distributed countdown timer

=> p * (1 – p)^k => Geometric distribution
Expected arrival time of a Bernoulli trial

When we sample at an instrumentation site
- Generate counter of instrumentation sites till next sample
- Using geometric distribution

At every instrumentation site
- Decrement counter
- Check if counter is 0
- If yes, sample

⇒Achieve "statistically fair" sampling without overhead of random number generation at each instrumentation site

20

Yet more tricks - instead of checking countdown every sample, use fast & slow paths

```
{
   if (countdown > 2) {
      /* fast path:  no sample ahead */
      countdown -= 2;
      p = p->next;
      total += sizes[i];
   } else {
      /* slow path:  sample is imminent */
      if (--countdown == 0) {
         check(p != NULL);
         countdown = getNextCountdown();
      }
      p = p->next;

      if (--countdown == 0) {
         check(i < max);
         countdown = getNextCountdown();
      }
      total += sizes[i];
   }
}
```

More to do to make it work for loops and procedure calls
Doubles memory footprint

21

Small benchmark programs

- bh
- bisort
- compress
- em3d
- health
- ijpeg
- mst
- perimeter
- power
- treeadd
- tsp
- vortex

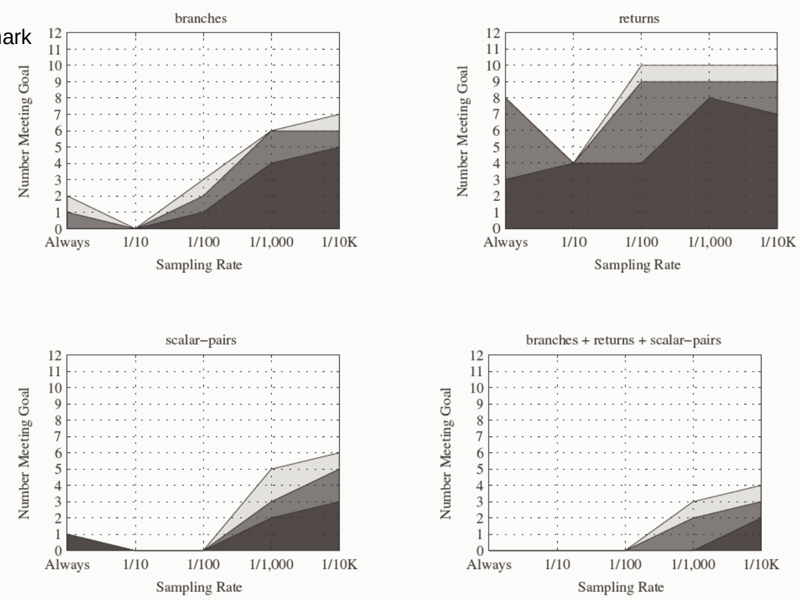


Figure 2.2: Benchmarks meeting performance goals in typical configuration: overhead ≤ 5% (■), overhead ≤ 10% (■), and overhead ≤ 15% (□)

Built a technique for sampling predicates cheaply!

How do we find bugs?

Statistical debugging

Predicate counters ->
bugs & bug predictors

23

There are several challenges from going from predicate counters to bugs and predictors

Feedback report R:
- (x > y) at line 33 of util.c 55 times
- … 100,000s more similar predicate counters

Label for report
- F – fail (e.g. it crashes), or S succeeds (e.g. it doesn't crash)

Challenges
- Lots of predicates – 100,000s
- Bug is deterministic with respect to program predicate
  - iff given predicate, bug must occur
    - predicate soundly predicts bug
  - Bugs may be nondeterministic & only occur sometimes
- All we have is sampled data
  - Even if a predicate deterministically predicts bug
  - We may not have sampled it on a particular run

=> Represent everything in probabilities rather than deterministic abstractions
- Instead of e.g. lattices, model checking state, Daikon true invariants, …

24

Notation

Uppercase variables denote sets; lower case denotes item in set

P – set of fundamental and inferred predicates

R – feedback report
- One bit – succeeded or failed
- Counter for each predicate p in P

R(p) – counter value for predicate p in feedback report R
- R(p) > 0 – saw predicate in run
- R(p) = 0 – never saw predicate in run

R(S) – counter value for instrumentation site S in feedback report R
- Sum of R(p) where p is sampled at S

B – bug profile – set of feedback reports caused by a single bug
- Failing runs may be in more than one bug profile if they have more than one bug

p is predictor iff R(p) > 0 ~> R in B
- Where ~> means statistically likely

Goal : find minimal subset A of P such that A predicts all bugs; rank importance of p in A
- Looking at this predicate will help you find a whole bunch of bugs!

Approach
- Prune away most predicates – totally irrelevant & worthless for any bug (98 – 99%) – really quickly
- Deal with other predicates in more detail

25

Deterministic bug example

Assume R(S) > 0 for all sites – i.e. all sites observed for all runs

R1: Succeeds (x > 5) at 3562 : R(P) = 23 (y > 23) at 1325 : R(P) = 0

R2: Fails (x > 5) at 3562 : R(P) = 13 (y > 23) at 1325: R(P) = 5

R3: Succeeds (x > 5) at 3562 : R(P) = 287 (y > 23) at 1325: R(P) = 0

Intuitively
- Which predicate is the best predictor?

26

Approach 1 – Eliminate candidate predicates using strategies

Universal falsehood
- R(P) = 0 on all runs R
- It is never the case that the predicate is true

Lack of failing coverage
- R(S) = 0 on all failed runs in R
- The site is never sampled on failed runs

Lack of failing example
- R(P) = 0 on all failed runs in R
- The predicate is not true whenever run fails

Successful counterexample
- R(P) > 0 on at least one successful run in R
- P can be true without causing failure
- (assumes deterministic bug)

=>Predictors should be true in failing runs and false in succeeding runs

27

Problems with Approach 1

Universal falsehood
- R(P) = 0 on all runs R
- It is never the case that the predicate is true

Lack of failing coverage
- R(S) = 0 on all failed runs in R
- The site is never sampled on failed runs

Lack of failing example
- R(P) = 0 on all failed runs in R
- The predicate is not true whenever run fails

Successful counterexample
- R(P) > 0 on at least one successful run in R
- P can be true without causing failure
- (assumes deterministic bug)

Assumes
- Only one bug
  - May be no deterministic predictor for all bugs
- At least one deterministic predictor of bug
  - Even a single counterexample will eliminate predicate
  - If no deterministic predictor, all predicates eliminated

28

Iterative bug isolation and elimination algorithm

1. Identify most important bug B
   - Infer which predicates correspond to which bugs
   - Rank predicates in importance

2. Fix B and repeat
   - Discard runs where R(p) > 0 for chosen predictor

2 increases the importance of predictors of less frequently bugs (occur in less runs)

Combination of assigning predicates to bugs and discarding runs handles multiple bugs!

29

How to find the cause of the most important bug?

Consider the probability that p being true implies failing run
Denote failing runs by Crash
Assume there is only a single bug (for the moment)

Fail(P) = Pr(Crash | P observed to be true)

Conditional probability
- Given that P happens, what's probability of crash

Can estimate Fail(P) for predicates
- Fail(P) = F(P) / (S(P) + F(P))
  - Count of failing runs / (Count of all runs)

- Not the true probability
  - it's a random variable we can never know
- But something that helps us best use observations to infer probability

30

What does Fail(P) mean?

Fail(P) = Pr(Crash | P observed to be true)

Fail(P) < 1.0
- Nondeterministic with respect to P
- Lower scores -> less predictive of bug

Fail(P) = 1.0
- Deterministic bug
- Predicate true -> bug!

31

But not quite enough….

```
f = ...;                  (a)
if (f == NULL) {          (b)
   x = 0;                 (c)
   *f;                    (d)
}
```

Consider
- Predicate (f == NULL) at (b)
  - Fail(f == NULL) = 1.0
  - Good predictor of bug!
- Predicate (x == 0) at (c)
  - Fail(x ==0) = 1.0 too!
    - S(X == 0) = 0, F(X ==0) > 0 if the bug is ever hit
  - Not very interesting!
    - Execution is already doomed when we hit this predicate
    - Bug has nothing to do with this predicate
    - Would really like a predicate that fails as soon as the execution goes wrong

32

Instead of Fail(P), what is the increase of P?

```
f = ...;                                                  (a)
if (f == NULL) {                                          (b)
   x = 0;                                                 (c)
   *f;                                                    (d)
}
```

Given that we've reached (c)
- How much difference does it make that (x == 0) is true?
- None – at (c), probability of crash is 1.0!

Fail(P) = Pr(Crash | P observed to be true)
- Estimate with
- Fail(P) = F(P) / (S(P) + F(P))

Context(P) = Pr(Crash | P observed at all)
- Estimate with
- Context(P) = F(P observed) / (S(P observed) + F(P observed)

33

Instead of Fail(P), what is the increase of P?

```
f = ...;                                                  (a)
if (f == NULL) {                                          (b)
   x = 0;                                                 (c)
   *f;                                                    (d)
}
```

Context(P) = Pr(Crash | P observed at all)
- Estimate with
- Context(P) = F(P observed) / (S(P observed) + F(P observed)

Increase(P) = Fail(P) – Context(P)
- How much does P being true increase the probability of failure vs. P being observed?
- Fail(x == 0) = Context(x == 0) = 1.0
- Increase(X == 0) = 1.0 – 1.0 = 0!

Increase(P) < 0 implies the predict isn't interesting and can be discarded
- Eliminates invaraints, unreachable statements, other uninteresting predicates
- Localizes bugs at where program goes wrong, not crash site
- So much more useful than Fail(P)!

34

Instead of Fail(P), what is the increase of P?

```
f = ...;                    (a)
if (f == NULL) {            (b)
   x = 0;                   (c)
   *f;                      (d)
}
```

Increase(P) = Fail(P) – Context(P)

But Increase(P) may be based on few observations
- Estimate may be unreliable

Use 95% confidence interval
- 95% chance that estimate falls within confidence interval
- Throw away predicates where this interval is not strictly above 0

35

Statistical interpretation of Increase(P) is likelihood ratio test

One of the most useful applications of statistics

Two hypotheses
1. Null Hypothesis: Fail(P) <= Context(P)
   - Alpha <= Beta
2. Alternative Hypothesis: Fail(P) > Context(P)
   - Alpha > Beta

Fail P and Context P are really just ratios
- Alpha = F(P) / F(P observed)
- Beta = S(P) / S(P observed)

LRT compares hypotheses taking into account uncertainty from number of observations

36

Thermometers diagrammatically illustrate these numbers

Length: log(# times P observed)

Context(P)	Lower bound on Increase(P) from confidence interval	Size of confidence interval	S(P)
How often true?	Minimally, how helpful?	How much uncertainty?	How many times is predicate true with no bug?
		Usuually small => tight interval	

37

Predicates true the most on failing runs

But also true a lot on nonfailing runs

Table 4.2: MOSS failure predictors sorted by F(*P*)

Thermometer	Context	Increase	S	F	F + S	Predicate
	0.176	0.007 ±0.012	22554	5045	27599	files[filesindex].language != 15
	0.176	0.007 ±0.012	22566	5045	27611	tmp == 0 is FALSE
	0.176	0.007 ±0.012	22571	5045	27616	strcmp != 0
	0.176	0.007 ±0.013	18894	4251	23145	tmp == 0 is FALSE
	0.176	0.007 ±0.013	18885	4240	23125	files[filesindex].language != 14
	0.176	0.008 ±0.013	17757	4007	21764	filesindex >= 25
	0.177	0.008 ±0.014	16453	3731	20184	M < M
	0.176	0.261 ±0.023	4800	3716	8516	config.winnowing_window_size != argc
	0.177	0.012 ±0.014	15325	3567	18892	i > 21
	0.131	0.018 ±0.012	17846	3125	20971	token_sequence[token_index].lineno <= token_index
	0.176	0.077 ±0.018	9136	3104	12240	tmp == 0 is FALSE
	0.176	0.077 ±0.018	9126	3095	12221	files[filesindex].language >= 14
	0.131	0.021 ±0.013	17256	3092	20348	token_index > lineno
	0.129	0.021 ±0.012	17589	3084	20673	i >= lineno
	0.131	0.023 ±0.013	16895	3060	19955	token_sequence[token_index].lineno < token_index
	0.176	0.020 ±0.015	12550	3056	15606	filesindex > 30
	0.115	0.008 ±0.011	21431	2985	24416	passage_index > yy_start
	0.176	0.021 ±0.016	11967	2935	14902	i > 24
	0.112	0.007 ±0.011	21403	2894	24297	min_index > yy_start
	0.110	0.007 ±0.011	21322	2826	24148	min_index > yy_start
	0.135	0.021 ±0.013	14493	2677	17170	fin > fin
	0.176	0.199 ±0.024	4332	2595	6927	config.winnowing_window_size < argc
	0.176	0.398 ±0.030	1924	2592	4516	i >= 8
	0.176	0.398 ±0.030	1924	2592	4516	i >= 6
	0.176	0.398 ±0.030	1924	2592	4516	i >= 2
	0.176	0.398 ±0.030	1924	2592	4516	i >= 4
	0.176	0.398 ±0.030	1924	2592	4516	i >= 9
	0.107	0.010 ±0.011	19385	2585	21970	(p + passage_index)->last_line <= filesindex
	0.107	0.006 ±0.011	20119	2578	22697	(p + passage_index)->last_line < last
	0.114	0.011 ±0.012	18046	2577	20623	(p + passage_index)->last_line <= filesindex
	0.107	0.011 ±0.011	19081	2568	21649	(p + passage_index)->last_line < filesindex
	0.107	0.007 ±0.011	19975	2568	22543	(p + passage_index)->first_line < fileid
	0.129	0.010 ±0.013	15906	2556	18462	i > 28
	0.114	0.012 ±0.012	17613	2551	20164	(p + passage_index)->last_line < filesindex
	0.116	0.027 ±0.013	15100	2517	17617	i > lineno
	0.116	0.025 ±0.013	15175	2506	17681	start > lineno
	0.114	0.008 ±0.012	17862	2491	20353	(p + passage_index)->first_line < fileid
	0.115	0.009 ±0.012	17554	2480	20034	(p + passage_index)->first_line < last
	0.107	0.064 ±0.014	11962	2464	14426	(p + passage_index)->first_line <= i

2701 additional predictors follow

Highest increase(P) (red bar) relative To total number of times Observed (length)

But they don't predict many bugs….

Table 4.3: MOSS failure predictors sorted by Increase(P)

Thermometer	Context	Increase	S	F	F + S	Predicate
	0.065	0.935 ± 0.019	0	23	23	((*(fi + i)))->this.last_token < filesbase
	0.065	0.935 ± 0.020	0	10	10	((*(fi + i)))->other.last_line == last
	0.071	0.929 ± 0.020	0	18	18	((*(fi + i)))->other.last_line == filesbase
	0.073	0.927 ± 0.020	0	10	10	((*(fi + i)))->other.last_line == yy_n_chars
	0.071	0.929 ± 0.028	0	19	19	bytes <= filesbase
	0.075	0.925 ± 0.022	0	14	14	((*(fi + i)))->other.first_line == 2
	0.076	0.924 ± 0.022	0	12	12	((*(fi + i)))->this.first_line < nid
	0.077	0.923 ± 0.023	0	10	10	((*(fi + i)))->other.last_line == yy_init
	0.080	0.920 ± 0.023	0	10	10	((*(fi + i)))->other.last_line == yy_n_chars
	0.080	0.920 ± 0.023	0	12	12	((*(fi + i)))->other.last_line == numlines_other
	0.081	0.919 ± 0.023	0	12	12	((*(fi + i)))->this.last_line == lineno
	0.082	0.918 ± 0.024	0	10	10	((*(fi + i)))->this.last_line <= yy_start
	0.082	0.918 ± 0.024	0	11	11	((*(fi + i)))->this.last_line == htmlscope
	0.085	0.915 ± 0.023	0	11	11	((*(fi + i)))->this.first_line < 7
	0.085	0.915 ± 0.024	0	10	10	((*(fi + i)))->other.last_line == lineno
	0.087	0.913 ± 0.025	0	12	12	((*(fi + i)))->other.last_line < yy_start
	0.087	0.913 ± 0.024	0	14	14	((*(fi + i)))->other.first_line < 3
	0.091	0.909 ± 0.025	0	10	10	((*(fi + i)))->this.last_line <= 1
	0.100	0.900 ± 0.009	0	526	526	yy_start == 17
	0.100	0.900 ± 0.009	0	503	503	yy_start > 16
	0.101	0.899 ± 0.009	0	522	522	yy_start > 11
	0.114	0.886 ± 0.007	0	66	66	last > 500
	0.115	0.885 ± 0.007	0	67	67	first > 500
	0.117	0.883 ± 0.011	0	79	79	((*(fi + i)))->other.last_token <= filesbase
	0.116	0.883 ± 0.012	1	769	770	((*(fi + i)))->other.last_line == yy_start
	0.118	0.882 ± 0.011	0	22	22	((*(fi + i)))->this.last_token == filesbase
	0.116	0.883 ± 0.012	1	774	775	((*(fi + i)))->this.last_line == 1
	0.116	0.883 ± 0.012	1	776	777	((*(fi + i)))->other.last_line == yyleng
	0.118	0.882 ± 0.011	0	13	13	((*(fi + i)))->other.last_line > pindex
	0.118	0.881 ± 0.012	1	769	770	((*(fi + i)))->this.last_line < 2
	0.118	0.881 ± 0.012	1	771	772	((*(fi + i)))->this.last_line == yy_start
	0.118	0.881 ± 0.012	1	772	773	((*(fi + i)))->other.last_line == 1
	0.118	0.880 ± 0.012	1	776	777	((*(fi + i)))->other.last_line < 2
	0.118	0.880 ± 0.013	2	772	774	((*(fi + i)))->this.last_line == yyleng
	0.124	0.876 ± 0.008	0	384	384	config.language == 17
	0.124	0.876 ± 0.008	0	391	391	config.language > 16
	0.117	0.876 ± 0.016	6	781	787	((*(fi + i)))->this.last_line == diff
	0.116	0.875 ± 0.017	7	784	791	((*(fi + i)))->other.last_line == diff
	0.131	0.869 ± 0.008	0	633	633	files[fileid].size < token_index

…… 2701 additional predictors follow ……

How do we rank bugs by importance?

Approach 1 – Importance(P) = Fail(P)

- # failing runs for which P is true
- Maximum soundness – find lots of bugs!
- May be true a lot on successful runs
  - Large white bands

Approach 2 – Importance(P) = Increase(P)

- How much does P true increase probability of failure?
  - Large red bands
- Maximum precision – very few false positives!
- Number of failing runs is small
  - Sub bug predictors – predict subset of a bug's set of failing runs
  - Large black bands

40

How do we balance precision and soundness in this analysis?

Information retrieval interpretation
- Recall / precision

Soundness = recall
- Match all the failing runs / bugs!

Preciseness = precision
- Don't match successful runs / no bug!

Information retrieval solution – harmonic mean

$$Importance(P) = \frac{2}{\frac{1}{Increase(P)} + \frac{1}{log(F(P))/log(NumF)}}$$

41

Table 4.4: MOSS failure predictors sorted by harmonic mean

Thermometer	Context	Increase	S	F	F + S	Predicate
	0.176	0.824 ± 0.009	0	1585	1585	files[filesindex].language > 16
	0.176	0.824 ± 0.009	0	1584	1584	strcmp > 0
	0.176	0.824 ± 0.009	0	1580	1580	strcmp == 0
	0.176	0.824 ± 0.009	0	1577	1577	files[filesindex].language == 17
	0.176	0.824 ± 0.009	0	1576	1576	tmp == 0 is TRUE
	0.176	0.824 ± 0.009	0	1573	1573	strcmp > 0
	0.116	0.883 ± 0.012	1	774	775	((*(fi + i)))->this.last_line == 1
	0.116	0.883 ± 0.012	1	776	777	((*(fi + i)))->other.last_line == yyleng
	0.111	0.832 ± 0.027	73	1203	1276	config.match_comment is TRUE
	0.116	0.883 ± 0.012	1	769	770	((*(fi + i)))->other.last_line == yy_start
	0.118	0.880 ± 0.012	1	776	777	((*(fi + i)))->other.last_line < 2
	0.118	0.881 ± 0.012	1	772	773	((*(fi + i)))->other.last_line == 1
	0.118	0.881 ± 0.012	1	771	772	((*(fi + i)))->this.last_line == yy_start
	0.118	0.881 ± 0.012	1	769	770	((*(fi + i)))->this.last_line < 2
	0.118	0.880 ± 0.013	2	772	774	((*(fi + i)))->this.last_line == yyleng
	0.117	0.876 ± 0.016	6	781	787	((*(fi + i)))->this.last_line == diff
	0.116	0.875 ± 0.017	7	784	791	((*(fi + i)))->other.last_line == diff
	0.115	0.866 ± 0.021	16	826	842	((*(fi + i)))->other.last_line <= 3
	0.117	0.855 ± 0.024	25	864	889	((*(fi + i)))->this.last_line <= 4
	0.131	0.810 ± 0.026	79	1258	1337	token_sequence[token_index].val >= 100
	0.118	0.863 ± 0.021	15	798	813	((*(fi + i)))->other.last_line <= 2
	0.118	0.865 ± 0.021	14	787	801	((*(fi + i)))->this.last_line <= 2
	0.116	0.851 ± 0.026	30	862	892	((*(fi + i)))->other.last_line <= 4
	0.118	0.855 ± 0.025	22	776	798	((*(fi + i)))->this.last_line == nextstate
	0.131	0.859 ± 0.016	7	711	718	token_index > 500
	0.131	0.869 ± 0.008	0	639	639	files[fileid].size < token_count
	0.119	0.849 ± 0.027	26	779	805	((*(fi + i)))->other.last_line == nextstate
	0.131	0.869 ± 0.008	0	633	633	files[fileid].size < token_index
	0.100	0.900 ± 0.009	0	526	526	yy_start == 17
	0.101	0.899 ± 0.009	0	522	522	yy_start > 11
	0.117	0.844 ± 0.028	32	794	826	config.match_comment is TRUE
	0.118	0.829 ± 0.031	49	876	925	((*(fi + i)))->this.last_line < nid
	0.115	0.796 ± 0.032	115	1171	1286	(p + passage_index)->last_line < 2
	0.100	0.900 ± 0.009	0	503	503	yy_start > 16
	0.117	0.828 ± 0.031	52	879	931	((*(fi + i)))->other.last_line < nid
	0.116	0.839 ± 0.030	37	794	831	((*(fi + i)))->other.last_line <= diff
	0.117	0.840 ± 0.030	36	788	824	((*(fi + i)))->this.last_line <= diff
	0.116	0.818 ± 0.033	65	914	979	((*(fi + i)))->this.last_line < 8
	0.118	0.833 ± 0.031	40	778	818	((*(fi + i)))->this.last_line <= nextstate

2701 additional predictors follow

42

Statistical Debugging Algorithm

1. Rank predicates by *Importance*.
2. Remove the top-ranked predicate P and discard all runs R (feedback reports) where $R(P) > 0$.
3. Repeat these steps until the set of runs is empty or the set of predicates is empty.

	Lines of Code	Runs		Sites	Predicate Counts		
		Successful	Failing		Initial	*Increase* > 0	Elimination
MOSS	6,001	26,299	5,598	35,223	202,998	2,740	21
CCRYPT	5,276	20,684	10,316	9,948	58,720	50	2
BC	14,288	23,198	7,802	50,171	298,482	147	2
RHYTHMBOX	56,484	12,530	19,431	145,176	857,384	537	15
EXIF	10,588	30,789	2,211	27,380	156,476	272	3

43

Questions

- How much better is this than release build asserts? How many of these predicates would never have been added as asserts?
- How much more useful are the predicates than just the bug stack? How much better do they localize the bug?

45