Statistical Debugging

ACM Dissertation Award (2005)

Thomas D. LaToza
17-654 Analysis of Software Artifacts

Despite the best QA efforts software will ship with bugs

Why would software be released with bugs?
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....

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?)
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

What’s wrong with debugging from a stack trace?

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

```c
main()
{
    // Extra methods!
    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();
}
```

CRASH HERE SOMETIMES

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

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
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

Deployment and Sampling
OSS users downloaded binaries submitting statistical debugging reports

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

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 ≥ y on line 319 of utils.c” would have been observed to be true 3 + 1 times,
“x ≠ y on line 319 of utils.c” would have been observed to be true 25 + 1 times, and
“x ≤ y on line 319 of utils.c” would have been observed to be true 25 + 3 times.
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

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
What's wrong with conventional sampling?

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

```c
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”)

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

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

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 $p = 1/5$

```
Example execution trace 0.0 0.0 0.0 1.0 0.0, 0.1, 0.1, 0.0, 0.0, 1.0...
```

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?
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 $p = 1/5$

What’s the probability that the next sample is at $t+k$?

Time $t$: $(1/5)$
Time $t+1$: $(4/5) \times (1/5)$
Time $t+2$: $(4/5)^2 \times (1/5)$
Time $t+3$: $(4/5)^3 \times (1/5)$
Time $t+k$: $(4/5)^k \times (1/5)$

$=> p \times (1 - p)^k =>$ Geometric distribution

Expected arrival time of a Bernoulli trial

Generate a geometrically distributed countdown timer

$=> p \times (1 - p)^k =>$ 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
Yet more tricks - instead of checking countdown every sample, use fast & slow paths

```c
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

---

Small benchmark programs

- bzip
- compress
- em3d
- health
- jpeg
- just
- perimeter
- power
- treeadd
- tsp
- vortex

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

Predicate counters -> bugs & bug predictors

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, …
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 \rightarrow R in B
  Where \rightarrow 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

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?
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

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
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!

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)

\[
\text{Fail}(P) = \Pr(\text{Crash} | P \text{ observed to be true})
\]

Conditional probability
Given that \( P \) happens, what’s probability of crash

Can estimate \( \text{Fail}(P) \) for predicates

\[
\text{Fail}(P) = \frac{F(P)}{S(P) + F(P)}
\]

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
What does $\text{Fail}(P)$ mean?

$\text{Fail}(P) = \Pr(\text{Crash} \mid \text{P observed to be true})$

$\text{Fail}(P) < 1.0$
- Nondeterministic with respect to $P$
- Lower scores -> less predictive of bug

$\text{Fail}(P) = 1.0$
- Deterministic bug
- Predicate true -> bug!

But not quite enough….

```
f = ...;
if (f == NULL) {
  x = 0;
  *f;   
}
```

Consider
- Predicate $(f == \text{NULL})$ at (b)
  - $\text{Fail}(f == \text{NULL}) = 1.0$
  - Good predictor of bug!

- Predicate $(x == 0)$ at (c)
  - $\text{Fail}(x == 0) = 1.0$ 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
Instead of \( \text{Fail}(P) \), what is the increase of \( P \)?

\[
\begin{align*}
f &= \ldots; \quad (a) \\
\text{if } (f == \text{NULL}) \{ \\
\quad x &= 0; \quad (c) \\
\quad *f; \quad (d)
\}
\end{align*}
\]

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!

\[
\begin{align*}
\text{Fail}(P) &= \Pr(\text{Crash} \mid P \text{ observed to be true}) \\
&= \frac{F(P)}{S(P) + F(P)} \\
\text{Context}(P) &= \Pr(\text{Crash} \mid P \text{ observed at all}) \\
&= \frac{F(P \text{ observed})}{S(P \text{ observed}) + F(P \text{ observed})}
\end{align*}
\]

\[
\begin{align*}
\text{Increase}(P) &= \text{Fail}(P) - \text{Context}(P) \\
\text{How much does } P \text{ being true increase the probability of failure vs. } P \text{ being observed?} \\
&= \text{Fail}(x == 0) - \text{Context}(x == 0) = 1.0 \\
&= 1.0 - 1.0 = 0!
\end{align*}
\]

Increase\((P) < 0\) implies the predict isn’t interesting and can be discarded
- Eliminates invarients, unreachable statements, other uninteresting predicates
- Localizes bugs at where program goes wrong, not crash site
- So much more useful than \( \text{Fail}(P) \)!
Instead of $\text{Fail}(P)$, what is the increase of $P$?

$\text{Increase}(P) = \text{Fail}(P) - \text{Context}(P)$

But $\text{Increase}(P)$ 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

Statistical interpretation of $\text{Increase}(P)$ is likelihood ratio test

One of the most useful applications of statistics

Two hypotheses
1. Null Hypothesis: $\text{Fail}(P) \leq \text{Context}(P)$
   $\alpha \leq \beta$
2. Alternative Hypothesis: $\text{Fail}(P) > \text{Context}(P)$
   $\alpha > \beta$

$\text{Fail}$ $P$ and $\text{Context}$ $P$ are really just ratios
$\alpha = \text{F}(P) / \text{F}(P \text{ observed})$
$\beta = \text{S}(P) / \text{S}(P \text{ observed})$

LRT compares hypotheses taking into account uncertainty from number of observations
Thermometers diagrammatically illustrate these numbers

Length: \( \log(\# \text{ times } P \text{ observed}) \)

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

Predicate true the most on failing runs

But also true a lot on nonfailing runs

Table 4.2: Moss failure predictors sorted by \( F(P) \)
Highest increase(P) (red bar) relative To total number of times Observed (length)

But they don’t predict many bugs….

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
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

\[
\text{Information retrieval solution} \quad = \quad \frac{1}{2} \left[ \frac{1}{\text{Increase}(F)} + \frac{1}{\log(F(P))/\log(\text{NumF})} \right]
\]

Table 4.4: Moss failure predictors sorted by harmonic mean
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.

<table>
<thead>
<tr>
<th>Lines of Code</th>
<th>Runs</th>
<th>Sites</th>
<th>Predicate Counts</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Successful</td>
<td>Failing</td>
<td>Sites</td>
</tr>
<tr>
<td>MOSS</td>
<td>6,001</td>
<td>26,299</td>
<td>5,598</td>
</tr>
<tr>
<td>CCrypt</td>
<td>5,276</td>
<td>20,684</td>
<td>10,316</td>
</tr>
<tr>
<td>BC</td>
<td>14,288</td>
<td>23,198</td>
<td>7,602</td>
</tr>
<tr>
<td>Rhythmobox</td>
<td>56,484</td>
<td>12,530</td>
<td>19,431</td>
</tr>
<tr>
<td>EXIF</td>
<td>10,588</td>
<td>30,789</td>
<td>2,211</td>
</tr>
</tbody>
</table>

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?