Statistical Debugging

OMS CS 6340

Motivation

• Bugs will escape in-house testing and analysis tools
  - Dynamic analysis (i.e. testing) is unsound
  - Static analysis is incomplete
  - Limited resources (time, money, people)
  - Software ships with unknown (and even known) bugs

An Idea: Statistical Debugging

• Monitor Deployed Code
  - Online: Collect information from user runs
  - Offline: Analyze information to find bugs
  - Effectively a "black box" for software

Benefits of Statistical Debugging

Actual runs are a vast resource!

• Crowdsourced Testing
  - Number of real runs >> number of testing runs
• Reality-directed Debugging
  - Real-world runs are the ones that matter most

Two Key Questions

• How do we get the data?
• What do we do with it?

Practical Challenges

1. Complex systems
   - Millions of lines of code
   - Mix of controlled and uncontrolled code
   - Threads
2. Remote monitoring constraints
   - Limited disk space, network bandwidth, power, etc.
3. Incomplete information
   - Limit performance overhead
   - Privacy and security
The Approach

- Guess behaviors that are “potentially interesting”
  - Compile-time instrumentation of program
- Collect sparse, fair subset of these behaviors
  - Generic sampling framework
- Feedback profile + outcome label (success vs. failure) for each run
- Analyze behavioral changes in successful vs. failing runs to find bugs
  - Statistical debugging

Overall Architecture

A Model of Behavior

- Assume any interesting behavior is expressible as a predicate \( P \)
  on a program state at a particular program point:
  Observation of behavior = observing \( P \)
- Instrument the program to observe each predicate
- Which predicates should we observe?

Branches Are Interesting

```
++branch_17[p!=0];
if (p) ...
else  ...
```

Track predicates:

```
branch_17:
p == 0
p != 0
```

Return Values Are Interesting

```
n = fopen(...);
++call_41[(n==0)+(n>=0)];
```

```
call_41:
n < 0
n > 0
n == 0
```

What Other Behaviors Are Interesting?

- Depends on the problem you wish to solve!

- Examples:
  - Number of times each loop runs
  - Scalar relationships between variables (e.g. \( i < j \))
  - Pointer relationships (e.g. \( p = q \) or \( p \neq \text{null} \))
Summarization and Reporting

- Feedback report per run is:
  - Vector of predicate states \( \bot, 0, 1, \top \)
  - Success/failure outcome label
- No time dimension, for good or ill

QUIZ (Part 1): Identify the Predicates

List all predicates tracked for this program, assuming only branches are potentially interesting:

```c
void main() {
  int z;
  for (int i = 0; i < 3; i++) {
    if (getc() == 'a')
      z = 0;
    else
      z = 1;
    assert(z == 1);
  }
}
```

QUIZ (Part 2): Populate the Predicates

Populate the predicate vectors and outcome labels for the two runs:

```c
void main() {
  int z;
  for (int i = 0; i < 3; i++) {
    if (getc() == 'a')
      z = 0;
    else
      z = 1;
    assert(z == 1);
  }
}
```

QUIZ (Part 1): Identify the Predicates

List all predicates tracked for this program, assuming only branches are potentially interesting:

- `getc() == 'a'`
- `getc() != 'a'`
- `i < 3`
- `i >= 3`

```
getc() == 'a'
getc() != 'a'
i < 3
i >= 3
```

Outcome label (S/F)

- \( \top \)
- \( \bot \)

The Need for Sampling

- Tracking all predicates is expensive
- Decide to examine or ignore each site...
  - Randomly
  - Independently
  - Dynamically
- Why?
  - Fairness
  - We need an accurate picture of rare events
A Naive Approach

- Toss a coin at each instrumentation site
- Too slow

```c
++count_42[p != NULL];
p = p->next;
++count_43[i < max];
total += sizes[i];
```

if (rand(100) == 0)
  ```c
  ++count_42[p != NULL];
p = p->next;
  ++count_43[i < max];
total += sizes[i];
  ```

Some Other Problematic Approaches

- Sample every kth predicate observed
  - Violates independence
  - Might miss predicates "out of phase"
- Use clock interrupt
  - No context
  - Not portable

Amortized Coin Tossing

- Observation: Samples are rare (e.g. 1/100)
- Idea: Amortize sampling cost by predicting time until next sample
- Implement as countdown values selected from geometric distribution
- Models inter-arrival time for biased coin toss
  - How many tails before next head?

```c
if (rand(100) == 0)
  ++count_42[p != NULL];
p = p->next;
++count_43[i < max];
total += sizes[i];
```

```c
if (countdown >= 2) {
  countdown -= 2;
p = p->next;
total += sizes[i];
} else {
  if (countdown-- == 0) {
    ++count_42[p != NULL];
countdown = next();
  }
p = p->next;
  if (countdown-- == 0) {
    ++count_43[i < max];
countdown = next();
  }
total += sizes[i];
}
```

Feedback Reports with Sampling

- Feedback report per run is:
  - Vector of sampled predicate states (⊥, 0, 1, ⊤)
  - Success/failure outcome label
  - Certain of what we did observe
    - But may miss some events
  - Given enough runs, samples ≈ reality
    - Common events seen most often
    - Rare events seen at proportionate rate

 QUIZ: Uncertainty Due to Sampling

Check all possible states that a predicate P might take due to sampling. The first column shows the actual state of P (without sampling).

<table>
<thead>
<tr>
<th>P</th>
<th>⊥</th>
<th>0</th>
<th>1</th>
<th>⊤</th>
</tr>
</thead>
<tbody>
<tr>
<td>⊥</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>⊤</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
QUIZ: Uncertainty Due to Sampling

Check all possible states that a predicate P might take due to sampling. The first column shows the actual state of P (without sampling).

<table>
<thead>
<tr>
<th>P</th>
<th>⊥</th>
<th>0</th>
<th>1</th>
<th>⊤</th>
</tr>
</thead>
<tbody>
<tr>
<td>⊥</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>⊤</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Overall Architecture Revisited

Finding Causes of Bugs

- We gather information about many predicates.
  - e.g., for bc (calculator program on Unix).
- Most of these are not predictive of anything.
- How do we find the few useful predicates?

Finding Causes of Bugs

How likely is failure when P is observed to be true?

\[ F(P) = \frac{\text{# failing runs where } P \text{ is observed to be true}}{\text{# failing runs}} \]

\[ S(P) = \frac{\text{# successful runs where } P \text{ is observed to be true}}{\text{# successful runs}} \]

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

Example: \( F(P) = 20, S(P) = 30 \) \( \Rightarrow \text{Failure}(P) = 20/50 = 0.4 \)

Finding Causes of Bugs

Tracking Failure Is Not Enough

Predicate \( x == 0 \) is an innocent bystander.
- Program is already doomed.

Context

What is the background chance of failure, regardless of P’s value?

\[ \text{F(P observed)} = \# \text{ failing runs observing } P \]

\[ \text{S(P observed)} = \# \text{ successful runs observing } P \]

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

Example: \( \text{F(P observed)} = 40, \text{S(P observed)} = 80 \) \( \Rightarrow \text{Context}(P) = 40/120 = 0.333... \)
A Useful Measure

Does the predicate being true increase the chance of failure over the background rate?

\[
\text{Increase(P)} = \text{Failure(P)} - \text{Context(P)}
\]

- A form of likelihood ratio testing
- \(\text{Increase(P)} = 1\) \(\Rightarrow\) high correlation with failing runs
- \(\text{Increase(P)} = -1\) \(\Rightarrow\) high correlation with successful runs
- \(P\) is an invariant (true for all runs) \(\Rightarrow\) \(\text{Increase(P)} = 0\)
- \(P\) is in dead code (never observed on any run) \(\Rightarrow\) \(\text{Increase(P)}\) undef.

Increase Works

<table>
<thead>
<tr>
<th>Case</th>
<th>Failure</th>
<th>Context</th>
<th>Increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>(f = \text{NULL})</td>
<td>0.67</td>
<td>1.00</td>
<td>0.33</td>
</tr>
<tr>
<td>(x = 0)</td>
<td>1.00</td>
<td>0.67</td>
<td>0.00</td>
</tr>
<tr>
<td>(i &lt; 3)</td>
<td>1.00</td>
<td>0.67</td>
<td>0.00</td>
</tr>
<tr>
<td>(i \geq 3)</td>
<td>0.00</td>
<td>0.67</td>
<td>-0.50</td>
</tr>
</tbody>
</table>

QUIZ: Computing Increase() Scores

<table>
<thead>
<tr>
<th>Case</th>
<th>Failure</th>
<th>Context</th>
<th>Increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{getc() == \text{'a'}})</td>
<td>1.00</td>
<td>0.50</td>
<td>0.50</td>
</tr>
<tr>
<td>(\text{getc() != \text{'a'}})</td>
<td>0.50</td>
<td>0.50</td>
<td>0.00</td>
</tr>
<tr>
<td>(i &lt; 3)</td>
<td>0.50</td>
<td>0.50</td>
<td>0.00</td>
</tr>
<tr>
<td>(i \geq 3)</td>
<td>0.00</td>
<td>0.50</td>
<td>-0.50</td>
</tr>
</tbody>
</table>

Isolating the Bug

<table>
<thead>
<tr>
<th>Case</th>
<th>Increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{getc() == \text{'a'}})</td>
<td>0.5</td>
</tr>
<tr>
<td>(\text{getc() != \text{'a'}})</td>
<td>0.0</td>
</tr>
<tr>
<td>(i &lt; 3)</td>
<td>0.0</td>
</tr>
<tr>
<td>(i \geq 3)</td>
<td>-0.5</td>
</tr>
</tbody>
</table>

void main() {
    int z;
    for (int i = 0; i < 3; i++) {
        if (\text{getc() == \text{'a'}})
            z = 0;
        else
            z = 1;
        assert(z == 1);
    }
}

A First Algorithm

1. Discard predicates having \(\text{Increase(P)} \leq 0\)
   - e.g., invariant bystander predicates, predicates correlated with success
   - Exact value is sensitive to small \(F(P)\)
   - Use lower bound of 95% confidence interval

2. Sort remaining predicates by \(\text{Increase(P)}\)
   - Again, use 95% lower bound
   - Likely causes with determinacy metrics
Isolating the Bug

<table>
<thead>
<tr>
<th>Condition</th>
<th>Increase</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getc() == 'a'</code></td>
<td>0.5</td>
</tr>
<tr>
<td><code>getc() != 'a'</code></td>
<td>0.0</td>
</tr>
<tr>
<td><code>i &lt; 3</code></td>
<td>0.0</td>
</tr>
<tr>
<td><code>i &gt;= 3</code></td>
<td>-0.5</td>
</tr>
</tbody>
</table>

```c
void main() {
    int z;
    for (int i = 0; i < 3; i++) {
        if (getc() == 'a')
            z = 0;
        else
            z = 1;
        assert(z == 1);
    }
}
```

Isolating a Single Bug in bc

```c
void more_arrays()
{
    ... /* Copy the old arrays. */
    for (indx = 1; indx < old_count; indx++)
        arrays[indx] = old_ary[indx];
    /* Initialize the new elements. */
    for (; indx < v_count; indx++)
        arrays[indx] = NULL;
    ...
}
```

It Works!

- At least for programs with a single bug
- Real programs typically have multiple, unknown bugs
- Redundancy in the predicate list is a major problem

Using the Information

- Multiple useful metrics: Increase(P), Failure(P), F(P), S(P)
- Organize all metrics in compact visual (bug thermometer)

Sample Report

Multiple Bugs: The Goal

Isolate the best predictor for each bug, with no prior knowledge of the number of bugs.
Multiple Bugs: Some Issues

- A bug may have many redundant predictors
  - Only need one
  - But would like to know correlated predictors
- Bugs occur on vastly different scales
  - Predictors for common bugs may dominate, hiding predictors of less common problems

An Idea

- Simulate the way humans fix bugs
- Find the first (most important) bug
- Fix it, and repeat

Revised Algorithm

Repeat
Step 1: Compute Increase(P), F(C), etc. for all predicates
Step 2: Rank the predicates
Step 3: Add the top-ranked predicate P to the result list
Step 4: Remove P and discard all runs where P is true
  - Simulates fixing the bug corresponding to P
  - Discard reduces rank of correlated predicates until no runs are left

Ranking by Increase(P)

<table>
<thead>
<tr>
<th>Thermometer</th>
<th>Context</th>
<th>Increase</th>
<th>S</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0.065</td>
<td>0.0019</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.065</td>
<td>0.0020</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.071</td>
<td>0.0020</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.073</td>
<td>0.0020</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.071</td>
<td>0.0028</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.075</td>
<td>0.0022</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.076</td>
<td>0.0022</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.077</td>
<td>0.0023</td>
<td>0</td>
</tr>
</tbody>
</table>

Problem: High Increase() scores but few failing runs!
- Sub-bug predictors cover special cases of more general problems

Ranking by F(P)

<table>
<thead>
<tr>
<th>Thermometer</th>
<th>Context</th>
<th>Increase</th>
<th>S</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.176</td>
<td>0.007 ± 0.012</td>
<td>22554</td>
<td>5045</td>
<td></td>
</tr>
<tr>
<td>0.176</td>
<td>0.007 ± 0.012</td>
<td>22566</td>
<td>5045</td>
<td></td>
</tr>
<tr>
<td>0.176</td>
<td>0.007 ± 0.012</td>
<td>22571</td>
<td>5045</td>
<td></td>
</tr>
<tr>
<td>0.176</td>
<td>0.007 ± 0.013</td>
<td>18894</td>
<td>4251</td>
<td></td>
</tr>
<tr>
<td>0.176</td>
<td>0.007 ± 0.013</td>
<td>18885</td>
<td>4240</td>
<td></td>
</tr>
<tr>
<td>0.176</td>
<td>0.008 ± 0.013</td>
<td>17757</td>
<td>4007</td>
<td></td>
</tr>
<tr>
<td>0.177</td>
<td>0.008 ± 0.014</td>
<td>16455</td>
<td>3731</td>
<td></td>
</tr>
<tr>
<td>0.176</td>
<td>0.261 ± 0.023</td>
<td>4800</td>
<td>3716</td>
<td></td>
</tr>
</tbody>
</table>

Problem: Many failing runs but low Increase() scores!
- Super-bug predictors cover several different bugs rather poorly

A Helpful Analogy

- In the language of information retrieval
  - Increase(P) has high precision, low recall
  - F(P) has a high recall, low precision
- Standard solution:
  - Take the harmonic mean of both
  - $2/(1/\text{Increase(P)} + 1/\text{F(P)})$
  - Rewards high scores in both dimensions
Sorting by the Harmonic Mean

<table>
<thead>
<tr>
<th>Thermometer</th>
<th>Context</th>
<th>Increase</th>
<th>S</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0.176</td>
<td>0.824 ± 0.009</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.176</td>
<td>0.824 ± 0.009</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.176</td>
<td>0.824 ± 0.009</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.176</td>
<td>0.824 ± 0.009</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.176</td>
<td>0.824 ± 0.009</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.176</td>
<td>0.824 ± 0.009</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.116</td>
<td>0.883 ± 0.012</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.116</td>
<td>0.883 ± 0.012</td>
<td>1</td>
</tr>
</tbody>
</table>

It works!

A Case Study: Exif

<table>
<thead>
<tr>
<th>Initial</th>
<th>Effective</th>
<th>Predicate</th>
</tr>
</thead>
<tbody>
<tr>
<td>i &lt; 0</td>
<td>maxlen &gt; 1900</td>
<td>o + a &gt; buf_size is TRUE</td>
</tr>
</tbody>
</table>

- Three predicates selected from among 156,476
- Each predicts a distinct crashing bug
- Found the bugs quickly using these predicates

What Have We Learned?

- Monitoring deployed code to find bugs
- Observing predicates as model of program behavior
- Sampling instrumentation framework
- Metrics to rank predicates by importance:
  - Failure(P), Context(P), Increase(P), ...
- Statistical debugging algorithm to isolate bugs

Key Takeaway

- A lot can be learned from actual executions
  - Users are executing them anyway
  - We should capture some of that information

- Crash reporting is a step in the right direction
  - But stack is useful for only about 50% of bugs
  - Doesn’t characterize successful runs
  - But this is changing ...

- Screenshot of permission to report crash
- Screenshot of permission to monitor all runs