Fault Localization Based on Natural Language Document Similarity

Zak Fry
Qualifying Exam Presentation
An Abundance of Bugs

• In 2005: “Every day almost 300 bugs appear that need triaging. This is far too much for the Mozilla programmers to handle”¹

• From 2003-2005 half of all bugs fixed in Mozilla still took longer than 29 days to resolve

• Over 10% of submitted Mozilla bugs during this time still have not changed status from “new” or “unverified”

Fault Localization

• Thus, we want to speed up the bug resolution process

• **Fault localization (FL):** the task of identifying the files that could be changed to address the defect.
State-of-the-Art Fault Localization

• Testing
  – Jones et al. – Tarantula: Coverage based on failed/passed cases

• Model Checking
  – Ball et al. – Coverage using SLAM counterexample and passing program runs

• Monitoring
  – Liblit et al. – Remote sampling and statistical bug isolation
Desired Solution Properties

• Use only limited static information
  – Require little additional developer effort
• Use lightweight, low overhead analysis
  – Time overhead consistent with current defect reporting rates
• Scale to large programs and diverse bug types
Eclipse Bug 91543:
“Exception when placing a breakpoint (double click on ruler).”
With M6 and also with build I20050414-1107 i get the stacktrace below now and then when wanting the place a breakpoint when double clicking in the editor bar. if i close the editor and reopen it again it goes ok.

!MESSAGE Error within Debug UI: !STACK 0
org.eclipse.jface.text.BadLocationException at
org.eclipse.jface.text.AbstractLineTracker.getLineInformation( AbstractLineTracker.java: 251) ...
Eclipse Bug 91543:
“Exception when placing a breakpoint (double click on ruler).”
With M6 and also with build I20050414-1107 i get the stacktrace below.
now and then when wanting the place a breakpoint when double clicking in the editor bar. if i close the editor and reopen it again it goes ok.

!MESSAGE Error within Debug UI: !STACK 0
org.eclipse.jface.text.BadLocationException
at
org.eclipse.jface.text.AbstractLineTracker.getLineInformation( AbstractLineTracker.java: 251) ...

Actual Change:
ToggleBreakpointAction
RulerToggleBreakpointAction

Eclipse Code Search:
Breakpoint
MethodBreakpointType
BreakpointsLocation
IChangeRulerColumn
BreakpointSpec
BreakpointEventImpl
MethodBreakpointSpec
RulerColumnBreakpoint
ClearAllBreakpoints
JavaExceptionBreakpoint
ValidBreakpointLocationLocator
TaskRulerAction
...
Eclipse Bug 91543: “Exception when placing a breakpoint (double click on ruler).”

With M6 and also with build I20050414-1107 i get the stacktrace below now and then when wanting the place a breakpoint when double clicking in the editor bar. if i close the editor and reopen it again it goes ok.

!MESSAGE Error within Debug UI: !STACK 0
org.eclipse.jface.text.BadLocationException
at org.eclipse.jface.text.AbstractLineTracker.getLineInformation( AbstractLineTracker.java: 251) ...
Insights

• Developers, Users, Maintainers use different structure and language to describe the same concepts
  – Natural language information is encoded in both defect reports and source code

• Word frequency and semantic language similarities present but not always obvious
Thesis

• Developers and users can experience and use the same concepts but will encode them differently (code vs. runtime behavior)
• We hypothesize that:
  – We can construct a model that exposes the mapping between the two types of documents such that we can locate faults at least as accurately as state-of-the-art tools on indicative benchmarks while limiting overhead
  – Our model’s success is explained by natural language information and not mitigating non-natural-language features
Proposed Algorithm Structure

• Input: Bug report(s) and the source code
• Basic Approach: develop specific document substructure similarity metrics that can be combined in a model to localize faults accurately
• Output: Ranked list of files in terms of likelihood of containing the bug in question
Methodology

• Structured Document Comparison
Feature Selection

• ANOVA of all possible document comparisons
  – For each bug: used all correct files and 150 distractors
  – Used only as a starting point – not a purely linear model

• Principle Component Analysis
  – Showed that a combination of 12 accounted for 99% of the overall variance in the data
  – Compared model using all features and one with the top 12 from the ANOVA to verify both steps
Feature Selection

• Final parameter space optimization via hill climbing
  – Started with ANOVA F-values and varied by +/- 10% each iteration
  – Stopped after 5 iterations when overall accuracy changed by less than 0.01%
## Model Features

<table>
<thead>
<tr>
<th>Report Substructure</th>
<th>Code Substructure</th>
<th>Relative Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>Report title</td>
<td>Method bodies</td>
<td>23.06</td>
</tr>
<tr>
<td>Report body</td>
<td>Method signatures</td>
<td>41.48</td>
</tr>
<tr>
<td>Report title</td>
<td>Comments</td>
<td>10.53</td>
</tr>
<tr>
<td>Report body</td>
<td>Class name</td>
<td>9.46</td>
</tr>
<tr>
<td>Report body</td>
<td>Comments</td>
<td>7.89</td>
</tr>
<tr>
<td>Stack trace</td>
<td>Class name</td>
<td>7.79</td>
</tr>
<tr>
<td>Report body</td>
<td>Method bodies</td>
<td>5.72</td>
</tr>
<tr>
<td>Component</td>
<td>Method bodies</td>
<td>4.30</td>
</tr>
<tr>
<td>Operating System</td>
<td>Comments</td>
<td>3.48</td>
</tr>
<tr>
<td>Component</td>
<td>Comments</td>
<td>3.03</td>
</tr>
<tr>
<td>Product</td>
<td>String literals</td>
<td>1.94</td>
</tr>
<tr>
<td>Report title</td>
<td>Method signatures</td>
<td>1.32</td>
</tr>
</tbody>
</table>
Evaluation

- Eclipse, Mozilla, and OpenOffice
  - 5345 confirmed bug reports with known fixes based on CVS check-in messages
- 48k files, 65mil LOC
  - Files totaling 6.5mil LOC involved in user fixes
- Closest previous work (Tarantula) used only 7 small programs comprising 2,557 LOC total
Score Metric

- Used by Jones et al. to compare against Cleve and Zeller, Renieris and Reiss
- Represents the number of files that can be eliminated from the overall search space
- Given our ranked list of 10,000 files and 1 faulty file: if the correct file appears as number 2,000:
  \[(10,000 – 2,000) / 10,000 = 80\%\]
Experiment 1 – “Score” Accuracy

• Compare against two logical baselines
  – Stack traces
  – Code churn

• Indirectly compare against the reported results of three state of the art tools
  – Incompatible benchmark requirements

• This work is successful if we are more accurate than the simple baselines and do comparably to existing tools while limiting overhead
# Experiment 1 - Results

## Table 1: Comparison of Defect Detection Techniques

<table>
<thead>
<tr>
<th>Set of Defects</th>
<th>Reports Used</th>
<th>Our Approach</th>
<th>Stack trace Baseline</th>
<th>Code churn Baseline</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reports from Openoffice</td>
<td>1040</td>
<td>67.9%</td>
<td>60.0%</td>
<td>72.8%</td>
</tr>
<tr>
<td>Reports from Eclipse</td>
<td>1272</td>
<td>86.9%</td>
<td>56.3%</td>
<td>73.1%</td>
</tr>
<tr>
<td>Reports from Mozilla</td>
<td>3033</td>
<td>92.2%</td>
<td>50.1%</td>
<td>93.9%</td>
</tr>
<tr>
<td>Reports with Stack traces</td>
<td>325</td>
<td>86.2%</td>
<td>65.1%</td>
<td>76.4%</td>
</tr>
<tr>
<td><strong>All reports</strong></td>
<td><strong>5345</strong></td>
<td><strong>88.2%</strong></td>
<td><strong>53.1%</strong></td>
<td><strong>84.8%</strong></td>
</tr>
</tbody>
</table>

## Table 2: Scores of Defect Detection Techniques

<table>
<thead>
<tr>
<th>Technique</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Renieris and Reiss</td>
<td>56.000%</td>
</tr>
<tr>
<td>Cleve and Zeller</td>
<td>63.415%</td>
</tr>
<tr>
<td>Jones et al. (Tarantula)</td>
<td>77.797%</td>
</tr>
<tr>
<td><strong>Our approach</strong></td>
<td><strong>88.193%</strong></td>
</tr>
</tbody>
</table>
Experiment 2: Natural Language Validation

- We measure the correlation of non-natural-language features and score.
- Then we systematically degrade the natural language information and measure performance degradation to confirm.
- This work is successful if correlations with extraneous features are low and performance degrades proportionally to word replacement.
Experiment 2 – Results

• All non-natural-language features had correlations with our Score measures of less than .15
• Degraded language by
  – Replace from same corpus
  – Replace from a dictionary
  – Replace with random characters
• In all cases, performance decreased monotonically
• Overall, word choice accounts for 10% Score
  – Explains some of the increased performance over both baselines and existing work
Performance

- Parsing all code after build releases
  - Under an hour for all projects
- Once indexed, read index files
  - Under a minute for all projects
- Create a ranked list for an incoming report
  - Under 10 seconds for all instances tested
- Expected use case: maximum of 1 min 10 sec per incoming report
Conclusion

• H1: We can build a static model that achieves higher score accuracy than existing tools when localizing faults on large indicative benchmarks

• H2: The model’s success is due to human chosen natural language inherent in defect reports and source code