Lecture 6
Fault Localization

ECE 422: Reliable and Secure Systems Design

UNIVERSITY OF ALBERTA

Instructor: An Ran Chen
Term: 2024 Winter

Schedule for today

- Key concepts from last class
- Fault localization
  - Traditional debugging
  - Spectrum-based technique
  - Information retrieval-based (IR-based) technique
- Deliverable

Fault removal

- Why? Despite fault tolerance efforts, not all faults are tolerated, so we need fault removal.
  - To keep in mind: fault tolerance is a must-have property for safety-critical systems.
  - But for software that we use everyday, we want to remove faults to improve user experience.
- Improving system dependability by:
  - Detecting existing faults through software verification and validation
  - Eliminating the detected faults

Fault removal as two concepts:

1. as a solution to improve availability, and thus dependability
2. as a solution for faults that affect user's daily activities (not tolerated)

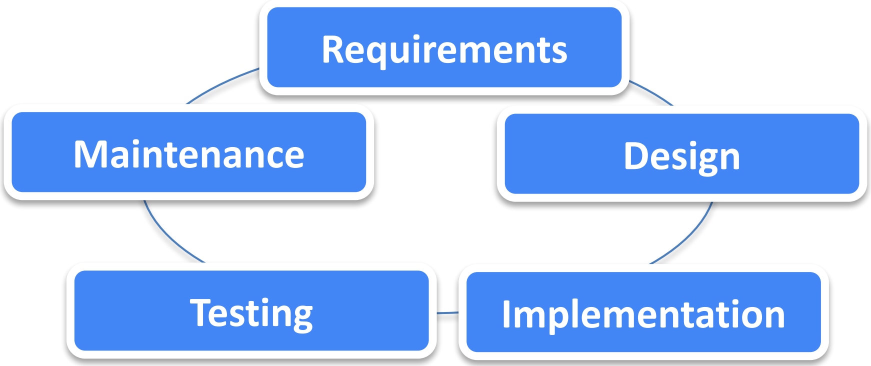
Dependability concept summary

- Dependability
  - Impairments
    - Faults: defect causing incorrect results
    - Errors: mistake in the code
    - Failures: occurs when system fails to perform its required function
  - Attributes
    - Availability: system is up and operational
    - Reliability: system performs without failure
  - Means
    - Fault removal: completely remove faults
    - Fault masking: hide faults, system still runs
    - Fault tolerance: prevent failures in presence of faults

Dependability in software development

Requirement phrase focus on the dependability attributes.

- Which dependability attributes are prioritized based on the user requirement?
  - Availability: to maximize operational time
  - E.g., the vending machine is always up and running
  - Reliability: to minimize system failures
  - E.g., the soda is never stuck in the vending machine



Dependability in software development

Design phrase focuses on the design properties.

- Do we need fault tolerance?
  - Fault tolerance as a design property
  - Fault tolerance is never integrated in the middle of the software development life cycle
  - Depend on the dependability attributes
- Which fault tolerance techniques to use?
  - Single vs multiple version fault tolerance techniques
  - Different consumption of resources
  - Multiple version techniques come with overheads for restoring the system state

Dependability in software development

Implementation phrase is about the actual implementation of the design.

- How to implement fault tolerance in the system?
  - Single vs multiple version fault tolerance techniques

Testing phrase ensures faults are removed from the system.

- What type of testing should we prioritize on?
  - Structural and functional testing
- How do we measure the effectiveness of structural testing?
  - Coverage analysis (e.g., path, branch, and statement coverage)

Dependability in software development

Maintenance phrase is about updating software to keep up with user requirements, including resolving faults in the system.

- Where is the fault?
- What are the root causes of the fault?

To answer these questions, developers use fault localization techniques.

Fault localization

Fault localization techniques have been proposed to assist in locating and understanding the root causes of faults.

Why? Fault localization as a debugging technique to maintain the system:

- Pinpoint the location to fix in the code
- Recover fast from bugs, and reduce its impact on the users
- Reduce manual debugging efforts, more time for new feature

What makes fault localization important?

Fault tolerance, masking and removal as high level solutions / guidelines to deal with faults. Fault localization (FL) provides a solution from the coding perspective to questions like:

- Where is the fault?
    - FL provides the exact location of faults at different granularity levels.
- Which part of the system is affected?
    - FL tries to detect all the fix locations.
- How can I reproduce the faults?
    - FL provides hints such as relevant test cases revealing the fault.
- … and more

Meta Research on software debugging

ICSE 2019

SapFix: Automated End-to-End Repair at Scale

Scaffle: Bug Localization on Millions of Files

Michael Pradel* — University of Stuttgart
Vijayaraghavan Murali — Facebook
Rebecca Qian — Facebook

ICSE 2021

TSE 2019

A Study of Bug Resolution Characteristics in Popular Programming Languages

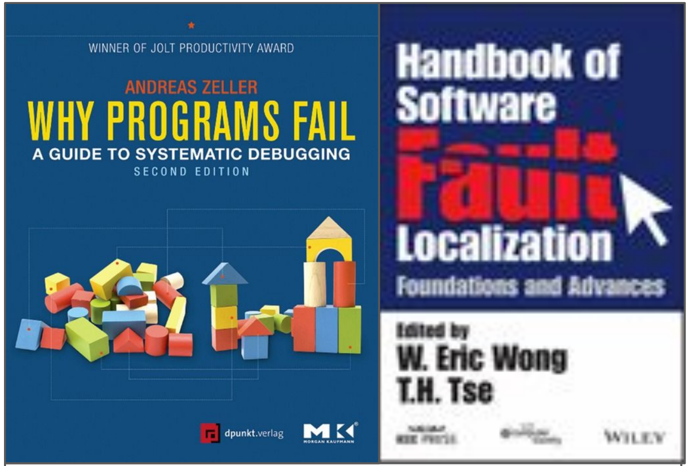
Jie M. Zhang*, Feng Li, Dan Hao, Meng Wang, Hao Tang, Lu Zhang, Mark Harman

ISSTA 2020

Industry-scale IR-based Bug Localization: A Perspective from Facebook

Vijayaraghavan Murali — Facebook, Inc.
Lee Gross — Facebook, Inc.
Rebecca Qian — Facebook, Inc.
Satish Chandra — Facebook, Inc.

Research community effort



The Debugging Book

Tools and Techniques for Automated Software Debugging

by Andreas Zeller

https://debuggingbook.org/, 2021

android studio

PyCharm

Visual Studio

eclipse

SafeRevert: When Can Breaking Changes be Automatically Reverted?
TAD Henderson, A Kondareddy, S Azad, E Nickell
hackthology.com - 3 days ago
PDF

Evolutionary Testing for Program Repair
H Ruan, HL Nguyen, R Shariffdeen, Y Noller, A Roychoudhury
Seed - 6 days ago
PDF

Automated Test Case Repair Using Language Models
AS Yaraghi, D Holden, N Kahani, L Briand
arXiv preprint arXiv:2401.06765 - 6 days ago
PDF

More articles from 6 days ago

Two-Level Information-Retrieval-Based Model for Bug Localization Based on Bug Reports
S Alsaedi, AAA Gad-Elrab, A Noaman, F Eassa
Electronics - 7 days ago
PDF

More articles from 7 days ago

DebugBench: Evaluating Debugging Capability of Large Language Models
R Tian, Y Ye, Y Qin, X Cong, Y Lin, Z Liu, M Sun
arXiv preprint arXiv:2401.04621 - 9 days ago
PDF

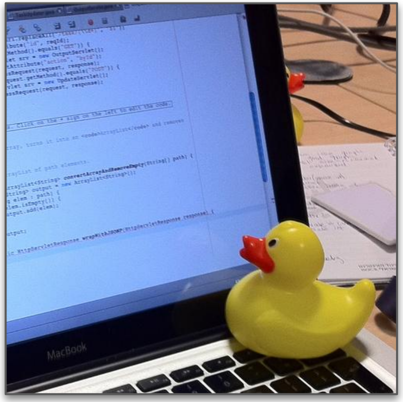
- Textbooks about software debugging alone
- Debuggers for IDEs
- New papers on FL or debugging everyday

Rubber duck debugging

Introduced by Dave Thomas and Andy Hunt (credited Andrew Hunt) in “The Pragmatic Programmer” book.

From a lists.ethernal.org post by Andy:

- Beg, borrow, steal, buy, fabricate, or otherwise obtain a rubber duck.
- Place rubber duck on desk and inform it you are just going to go over some code with it, if that’s all right.
- Explain to the duck what your code does line by line.
- At some point, you will realize what you are telling to the duck is not in fact what you are actually doing.



If you don’t have a rubber duck, a co-worker works too.

Traditional debugging

Traditional debugging methods presents guides or techniques for findings faults manually.

However, there are many challenges:

- Searching process is time-consuming
- Challenging to understand the system as it evolves to be more complex
- Some faults can be time-sensitive, putting more pressures on developers

Fault localization

Fault localization present automated techniques for locating the faults in the source code.

There are many families of fault localization techniques:

- Spectrum-based techniques
- Information retrieval-based techniques
- Mutation-based techniques (not covered in class)
- Historical-based techniques (not covered in class)
- and more …

Spectrum-based fault localization

Spectrum-based fault localization (SBFL), also known as statistical debugging, uses the results of test cases to identify the location of faults.

- Pinpoints the most suspicious program element (e.g., statement, method, file) based on the code coverage.
- Basic intuition: the location of code that is covered by more failing tests and less passing tests are more likely to contain faults.

Step 1 - Run all tests

Step 1: Run all tests

- Collect test results (passed or failed)
- Collect the code coverage (statement coverage)

	T_1
S_1	✓
S_2	
S_3	
S_4	✓

For example

- T1 is a passing test.
- T1 covers statement 1 and 4.

Step 2 - Build test execution profiles

Step 2: Build test execution profiles

- For every executable statement in the code, collect the tests that executed that statement
- For example:
    - Statement S1 was executed by one passing test, T1
    - Statement S2 was executed by one failing test, T2
    - Statement S3 was executed by two failing tests, T2 and T3
    - Statement S4 was executed by one passing test, T1 and one failing test, T2

	T_1	T_2	T_3
S_1	✓		
S_2		✓	
S_3		✓	✓
S_4	✓	✓	

Execution profile

Step 3: Calculate the suspiciousness score

Step 3: Calculate the suspiciousness score

- Use SBFL formulas to calculate a suspiciousness score for *each* program element
- Example of SBFL formula: Ochiai formula

$$Ochiai(element) = \frac{e_f}{\sqrt{(e_f + n_f) \cdot (e_f + e_p)}}$$

e_f	Number of failed tests that execute the program element.
e_p	Number of passed tests that execute the program element.
n_f	Number of failed tests that do not execute the program element.
n_p	Number of passed tests that do not execute the program element.

Step 3: Calculate the suspiciousness score

- For example

	T_1	T_2	T_3
S_3		✓	✓

$$\frac{e_f}{\sqrt{(e_f + n_f) \cdot (e_f + e_p)}}$$

→

$e_f = 2$
$n_f = 0$
$e_p = 0$

→

$$\frac{2}{\sqrt{(2+0) * (2+0))}}$$

↓

Suspiciousness score = 1

e_f	Number of failed tests that execute the program element.
e_p	Number of passed tests that execute the program element.
n_f	Number of failed tests that do not execute the program element.
n_p	Number of passed tests that do not execute the program element.

Step 3: Calculate the suspiciousness score

	T_1	T_2	T_3
S_1	✓		
S_2		✓	
S_3		✓	✓
S_4	✓	✓	

Execution profiles

- $S_1 = \dfrac{e_f}{\sqrt{(e_f + n_f) \cdot (e_f + e_p)}} = 0$
- $S_2 = \dfrac{1}{\sqrt{(1+1) * (1+0)}} = 0.71$

…

Statement	Suspiciousness score
S_1	0.00
S_2	0.71
S_3	1.00
S_4	0.50

Step 4: Rank elements by suspiciousness

Step 4: Rank the program elements by their suspiciousness score

- Output: a ranked list of suspicious elements is provided to the developer for manual bug fix.
- The element with the higher suspiciousness score is more likely to contain the fault.

Spectrum-based fault localization

Source file **A**

Run tests

Failed test

	T_1	T_2	T_3
S_1	✓		
S_2		✓	
S_3		✓	✓
S_4	✓	✓	

Execution profiles

SBFL

Statement	Suspiciousness score
S_1	0.00
S_2	0.71
S_3	1.00
S_4	0.50

Hint: program elements that are covered by more failing tests but less passing tests are more suspicious.

Limitations of SBFL

- Require the code coverage information, which may not always be available.
- Possible tie issue: same score is given to the elements covered by the equal number of passing and failing tests.
- Assume that test failures are related to the fault. Not the case for flaky tests.

Information retrieval-based fault localization

Information retrieval-based fault localization (IR-based FL) uses the textual description in the bug reports to locate the fault.

- Pinpoints the most suspicious program element (e.g., statement, method, file) based on the textual similarity between the bug description and the source code.
- Basic intuition: the description in the bug report and the faulty program element are likely to share the same tokens (words)

Step 1 - Preprocessing

Step 1.1 - Preprocess the bug report

- Text normalization: transforming text into a single canonical form
  - E.g., convert “stopwords”, “stop words”, “stop-words” to just “stopwords”
- Stopword removal: removing common, non-meaningful words
  - E.g., remove “is”, “a”
- Stemming: reducing text to their base form
  - E.g., convert “singing”, “sing”, “sung” to “sing”

Step 1 - Preprocessing

Step 1.2 - Preprocess the source code

- Keyword removal: removing programming language specific keywords
  - E.g., remove “for”, “if” for Java
- Concatenated words splitting
  - E.g., convert “getAverage” to “get” and “average”

Step 2 - Build vector space model

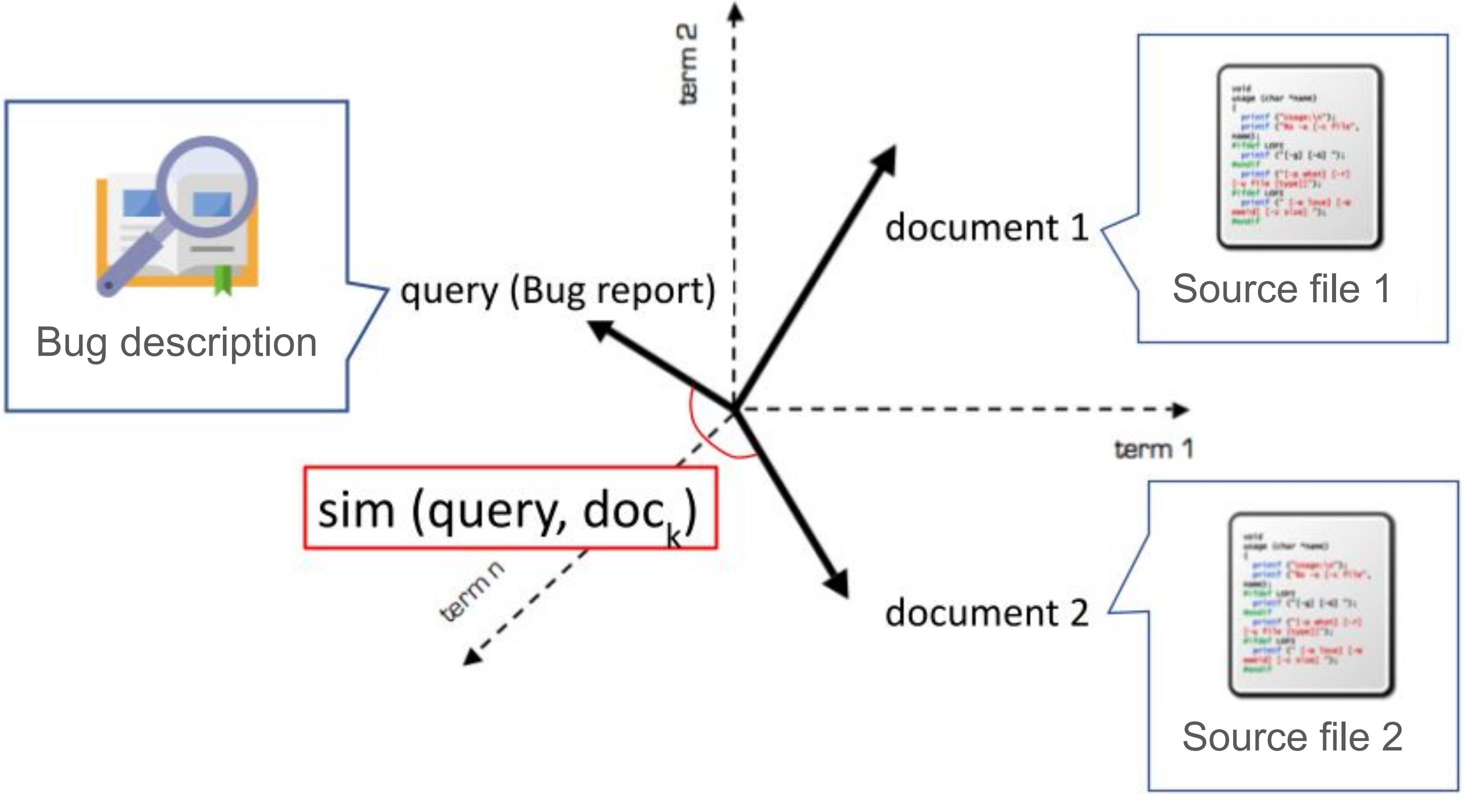
Step 2 - Build vector space model

- Vector space model: representing text documents as vectors so that we can calculate the similarity between vectors
- Intuition: transforms fault localization to a search problem

Example of search problem:

- Search query: bug description
- Documents: source code files
- Find the document with the highest similarity score to our search query
- Documents with the highest similarity are more likely to contain faults.

Vector representations



Example of vector representation

Suppose that:

- Bug report/query = “a problem with the classNotFound exception.”
- Source file/document = “ get classNotFound exception return exception”

	a	problem	with	the	classNotFound	exception	get	return
A	1	1	1	1	1	1	0	0
B	0	0	0	0	1	2	1	1

Vector representation:

- A = [1, 1, 1, 1, 1, 1, 0, 0]
- B = [0, 0, 0, 0, 1, 2, 1, 1]

Step 3 - Calculate the similarity metrics

Step 3 - Calculate the similarity metrics

- The similarity metric (e.g., cosine similarity) is based on the angle between the two vectors:

$$\cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\| \|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \cdot \sqrt{\sum_{i=1}^{n} B_i^2}}$$

Limitations of IR-based FL

- Assume high quality of bug reports
- In reality, there is always back-and-forth communication between the developer and the users.
- Only leverages the “visible” information in bug reports
- Useful debugging hints are often attached as error logs, screenshot, or even test cases as part of the bug report.

Deliverable

Due next Wednesday midnight, the grading scheme (total of 5 points):

- A class diagram (1 point)
- A section describing tools and technologies (1 point)
- Two user stories (2 points)
- A timeline showing your planning of the sub-tasks (1 point)

Cosine similarity

Cosine similarity is calculated by the equation:

$$\cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\| \|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \cdot \sqrt{\sum_{i=1}^{n} B_i^2}}$$

Suppose that our goal is to calculate the similarity of a bug report below:

- Bug report/query = “a problem with the classNotFound exception.”
- Source file/document = “ get classNotFound exception return exception”

Cosine similarity

Step 1: create a vector representation of the query and document.

- Bug report/query = “a problem with the classNotFound exception.”
- Source file/document = “ get classNotFound exception return exception”

	a	problem	with	the	classNotFound	exception	get	return
A	1	1	1	1	1	1	0	0
B	0	0	0	0	1	2	1	1

Vector representation:

- A = [1, 1, 1, 1, 1, 1, 0, 0]
- B = [0, 0, 0, 0, 1, 2, 1, 1]

Cosine similarity

Step 2: calculate the dot product and magnitude of these vectors

Vector representation:

- A = [1, 1, 1, 1, 1, 1, 0, 0]
- B = [0, 0, 0, 0, 1, 2, 1, 1]

Dot product of the vectors:

$A * B = 1 \times 0 + 1 \times 0 + 1 \times 0 + 1 \times 0 + 1 \times 1 + 1 \times 2 + 0 \times 1 + 0 \times 1 = 3$

Magnitude of the vectors:

$|| A || = \sqrt{} (1^2 + 1^2 + 1^2 + 1^2 + 1^2 + 1^2 + 0 + 0) = \sqrt{} 6$

$|| B || = \sqrt{} (0^2 + 0^2 + 0^2 + 0^2 + 1^2 + 2^2 + 1^2 + 1^2) = \sqrt{} 5$

Cosine similarity

Step 3: calculate the cosine similarity

$$similarity(A, B) = \frac{A * B}{\|A\| \, \|B\|} = \frac{3}{\sqrt{6} * \sqrt{5}} = 0.5477$$

The bug report and source code file could be said to be 55% similar.