Institute of Software Technology

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelor Thesis Nr. 129

Systematic Architecture Level Fault Diagnosis Using Statistical Techniques

Fabian Keller

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr. Lars Grunske
Supervisor:	Prof. Dr. Lars Grunske
Commenced:	May 6, 2014
Completed:	November 3, 2014
CR-Classification:	D.2.5

Abstract

In the past various spectrum-based fault localization (SBFL) algorithms have been developed to pinpoint a fault location given a set of failing and passing test executions. Most of the algorithms use similarity coefficients and have only been evaluated on established benchmark programs like the Siemens set or the space program from the Software-artifact Infrastructure Repository. In addition to that, SBFL has not been applied by developers in practice yet. This study evaluates the feasibility of applying SBFL to a real-world project, namely AspectJ. From an initial set of 110 manually classified faulty versions, a maximum of seven bugs can be found after examining the 1000 most suspicious lines produced by various SBFL techniques. To explain the result, the influence of the program size is examined using different metrics and evaluations. In general, the program size has a slight influence on some metrics, but is not the primary explanation for the results. The results seem to originate from the metrics currently used throughout the research community to assess SBFL performance. The study showcases the limitations of SBFL with the help of different performance metrics and the insights learned during manual classification. Moreover, additional performance metrics that are better suited to evaluate the fault localization performance are proposed.

Zusammenfassung

In der Vergangenheit wurden verschiedene Algorithmen zur spektrumsbasierten Fehlerlokalisierung (SBFL) entwickelt, um eine fehlerhafte Stelle mithilfe einer Menge von fehlschlagenden und bestandenen Testfällen bestimmen zu können. Die meisten Algorithmen nutzen dabei Ähnlichkeitskoeffizienten und wurden lediglich mit etablierten Vergleichsprogrammen, wie dem “Siemens set” oder dem “space program” aus dem Software-artifact Infrastructure Repository evaluiert. Darüber hinaus wurde SBFL noch nicht von Entwicklern in der Praxis integriert. Diese Studienarbeit untersucht wie sich SBFL auf ein echtes Projekt namens AspectJ anwenden lässt. Aus einer ursprünglichen Menge von 110 manuell klassifizierten fehlerhaften Versionen konnten nach dem Untersuchen der 1000 verdächtigsten Zeilen von verschiedenen Algorithmen, nur sieben Fehler gefunden werden. Um die Ergebnisse erklären zu können, wird der Einfluss der Programmgröße durch verschiedene Metriken untersucht. Im Allgemeinen hat die Programmgröße einen kleinen Einfluss auf einige Metriken, allerdings ist dies nicht die hauptsächliche Ursache für die Ergebnisse. Diese scheinen eher durch die Metriken, die aktuell von der Wissenschaft verwendet werden, um die Leistung eines SBFL-Algorithmus zu beurteilen, verursacht zu sein. Diese Studienarbeit zeigt die Einschränkungen von SBFL mit Hilfe von verschiedenen Leistungs-Metriken und den Erkenntnissen der manuellen Klassifizierung auf. Darüber hinaus werden zusätzliche Leistungs-Metriken, die besser zur Bewertung der Fehlerlokalisierungsleistung geeignet sind, vorgeschlagen.

Contents

Chapter 1

Introduction

The market for software development is continuously growing with software systems becoming increasingly complex and interwoven. However, increasingly complex systems also increase the likelihood of software faults that have to be diagnosed and then fixed. This debugging process commonly is a manual and iterative task where a developer first has to choose a suspicious piece of code that might be the fault and then verify whether it actually is the fault or not. If it is not faulty, the procedure is repeated until the fault location has been found and fixed. According to Britton et al. [2013] the global costs of software debugging for the year 2013 has risen to USD 312 billion. This is approximately one fourth of the total global costs of software development. In order to reduce the costs for software development it is of high importance to improve the effectiveness and efficiency of the debugging process. The effectiveness can be improved by helping the developer to find the root cause of the failure whereas the efficiency can be improved by helping the developer to find the root cause with less effort and time.

With an improved debugging process not only the cost of software development can be reduced, a faster debugging process allows faster release cycles that are less error-prone. Especially software reliability can greatly be improved by reducing the mean time to repair. Besides the improvements from a project perspective, an improved debugging process also prevents developers from tedious debugging tasks.

In order to help developers locate a fault more efficiently spectrum-based fault localization (SBFL) [Jones et al., 2002] has been proposed. In a nutshell, SBFL assigns a suspiciousness score to program elements that are computed by the number of failing and passing test cases where the program element can either be involved or not involved. Intuitively, program elements that are involved in failing executions or program elements that are not involved in successful executions are more likely to contain the fault. A developer may then inspect the most suspicious program elements until the fault is found.

Most research in this area has been validated with several projects from the Software-artifact Infrastructure Repository (SIR) [Do et al., 2005]. It is generally assumed that the evidence found with the SIR can be applied to larger projects. The first goal of this study is to investigate how the program size of a large open-source project influences SBFL performance. To measure SBFL performance, several performance metrics are commonly used throughout the research community. Though a recent human study by Parnin and Orso [2011] has shown that the assumptions the performance metrics are based on do not hold in practice. The second goal of this study it to explore additional SBFL performance

metrics that are capable of revealing limitations that particularly occur when applying SBFL to large projects.

The next chapter presents relevant foundations and technologies. Then, Chapter 3 details the applied research methods and Chapter 4 explains the experiment setup of the AspectJ case study. In Chapter 5 the results and their implications are analyzed. Finally, Chapter 6 concludes the study and outlines future work.

Chapter 2

Fault Localization

Software is usually developed to meet certain requirements. If software development was done perfectly, the software would fulfill the requirements in any case and would never fail. However, software systems begin to grow in complexity very fast and thus sooner or later have failing executions that do not meet the requirements. Those failing executions are denoted as bugs and are reproducible, even if it might be difficult to recreate the circumstances under which the software failed.

The occurred failure has a root cause in the source code which is henceforth denoted as fault location or simply fault. In order to fix the failure, a software developer first has to locate the fault and then fix it. As software projects become larger and older a rising fraction of the total costs of a project can be composed by fixing bugs. The costs rise with the time developers need to locate a fault, as locating a fault requires project knowledge, programming language expertise and debugging experience.

As the debugging process is a longsome and thus cost intensive process, a lot of effort has been put into improving this process. Several approaches (see Wong and Debroy [2010] for a comprehensive overview or Jones [2008] for a detailed overview and comparison) exist to assist a developer in locating the fault location of an occurring failure. This chapter outlines the various techniques available and puts this work into the context.

2.1 Traditional Fault Localization

The first traditional approach used by developers to debug is simply to print additional information on the screen in order to provide the developer with insight into the application state and execution flow. This debugging approach is very intuitive to developers and broadly used. The developer has to decide where to add the print statements in the application code and what variables they should print to the screen. The goal is to place the print statements such that the output helps the developer to verify whether the inspected piece of code works as expected or not. By repeating this procedure the developer gains knowledge of the execution flow of the program and is then able to manually pinpoint the fault location and correct the execution flow.
Adding print statements is time intensive and adds the risk that the developer forgets to remove the debugging modifications before taking the code to production. To tackle this, a commonly used and improved technique is a symbolic debugger which allows the

developer to halt the program execution at a specified point (the breakpoint) and then step through the program execution statement by statement. Symbolic debuggers are typically part of an integrated development environment such as Visual Studio[1] or Eclipse[2]. While stepping through the code, the developer is able to inspect the value of all available variables in the current scope and may even alter the position of the breakpoints. The main advantage over adding print statements is that the developer does not have to decide which variables to inspect upfront, but may choose to effortlessly inspect multiple variables of even complex types, like nested objects or list structures. However, it still requires domain knowledge, experience, multiple iterations and smart observation to place the breakpoints at the correct location, which is what also makes this debugging technique very expensive.

2.2 Advanced Fault Localization

The main disadvantage of traditional fault localization techniques is that developers have to decide for themselves on which parts of the program to debug. Developers usually decide where to debug using experience and intuition and thus the decision is very error-prone.

In order to help a developer to focus on fewer parts of the program Weiser [1981] proposed a slicing technique aiming to guide the developer to important lines while debugging. If a developer wants to know which statements influence the value of a certain variable the slicing technique can point the developer to those lines in the program that may have an influence on the state.

A major disadvantage of the proposed slicing technique is that the extracted program slice is very large in real-world scenarios as it contains all lines of code that may influence the value of the variable for all possible inputs, whereas the actual error most probably occurs only when executing a specific input involving only a subset of the lines in the slice. To reduce the size of the extracted slice Korel and Laski [1988] have proposed *dynamic slicing*. The key idea behind dynamic slicing is to only include lines that were involved in a particular execution. Applying this technique to failing executions extracts a slice that must contain the fault location and thus helps developers to focus on a smaller part of the program. However, even dynamic slicing techniques still extract large slices and have not been applied by developers in practice.

2.3 Spectrum-based Fault Localization

Spectrum-based fault localization (SBFL) can be used to locate faults in a very generic system model. In general, all systems that consist of multiple components and provide the ability to track the involvement of single components in passing or failing executions of the system, are capable of leveraging SBFL to diagnose a faulty component in case of an

[1] http://www.visualstudio.com
[2] http://www.eclipse.org/ide

error. The input for SBFL is a set of executions with each execution being either classified as passing execution, i.e. no error occurred, or as a failing execution, i.e. an error occurred. For each execution SBFL needs to know which components are involved or not involved in the specific execution. A single execution with its involvement data is generally referred to as a passing/failing *spectra* and a set of spectra is used as input for SBFL. A set of spectra can be represented using a $M \times N$ binary matrix:

$$\begin{array}{c} \\ \\ \\ M \text{ spectra} \end{array} \begin{array}{c} \\ N \text{ components} \\ \begin{array}{cccc} C_1 & C_2 & \dots & C_N \end{array} \\ \begin{array}{c} S_1 \\ S_2 \\ \vdots \\ S_M \end{array} \begin{bmatrix} h_{11} & h_{12} & \dots & h_{1N} \\ h_{21} & h_{22} & \dots & h_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ h_{M1} & h_{M2} & \dots & h_{MN} \end{bmatrix} \end{array} \begin{array}{c} \text{error} \\ \text{vector} \\ \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_M \end{bmatrix} \end{array} \tag{2.1}$$

Every row i with $i \in \{1..M\}$ of the matrix represents the spectra S_i. If the spectra originates from a passing execution then $e_i = 0$, whereas $e_i = 1$ for a failed execution. Every matrix cell h_{ij} with $i \in \{1..M\}, j \in \{1..N\}$ contains a 0 if the component C_j was not executed in spectra S_i and a 1 if the component was executed in the spectra.

In general, SBFL can be applied to various problem domains with different abstraction levels. However, in this study the focus is put on applying SBFL to real-world software development projects to assist developers when debugging. The following subsections introduce the general technique used to locate faults using SBFL, but without loss of generality terminology and examples found in software systems are used to explain the technique.

2.3.1 Locating Faults Using SBFL

In software systems the faulty components that need to be localized can be of any abstraction level, e.g. statements, basic blocks, methods, classes or packages. Current research primarily focuses on the statement or block level, but there are studies examining the capabilities of SBFL for other abstraction levels such as embedded systems [Abreu et al., 2009a] or distributed systems [Casanova et al., 2011]. To obtain the involvement of low-level software components like statements or blocks, various spectra types, such as branch count/hit spectra, path count/hit spectra or data-dependence count/hit spectra [Harrold et al., 2000], can be used. A "count" spectra counts how many times a specific spectra element was executed (numeric value), whereas a "hit" spectra only reveals whether a spectra element was executed at least once or never (boolean value). Statement hit/count spectra are easy to obtain with code coverage tools and contain sufficient data for SBFL to localize a fault. According to Harrold et al. [2000] all spectra types correlate with the exposure of faults and thus can be used to locate faults.

To locate a fault, SBFL assigns a suspiciousness score to each spectra element. The

higher the suspiciousness score of a spectra element, the more likely it is that the element contains the actual fault. The suspiciousness score is calculated by counting the number of passed/failed involvements/non-involvements of each spectra element and combining these four numbers using a formula, which is henceforth call *ranking metric*. Adapting Abreu et al. [2006], the four numbers are denoted as $\langle n_{np}, n_{nf}, n_{ip}, n_{if} \rangle$ where the first index represents the involvement (i) or non-involvement (n) of the spectra element and the second index represents passing (p) or failing (f) executions. Using the matrix notation of Eq. (2.1) the four numbers can be formally defined as:

$$\begin{aligned} n_{np}(C_j) &= \left|\{h_{ij} \,|\, h_{ij} = 0 \wedge e_i = 0\}\right| \\ n_{nf}(C_j) &= \left|\{h_{ij} \,|\, h_{ij} = 0 \wedge e_i = 1\}\right| \\ n_{ip}(C_j) &= \left|\{h_{ij} \,|\, h_{ij} = 1 \wedge e_i = 0\}\right| \\ n_{if}(C_j) &= \left|\{h_{ij} \,|\, h_{ij} = 1 \wedge e_i = 1\}\right| \end{aligned} \tag{2.2}$$

After a suspiciousness score has been assigned to each spectra element using a ranking metric, the statements can be ranked in descending order by their suspiciousness. Current research assumes that a developer can then find the fault by going through the list element by element, starting with the most suspicious element. However, Parnin and Orso [2011] have shown in a case study that developers in practice do not follow the list in a strict order, but rather traverse the list by skipping elements and changing the direction of going up and down the list multiple times.

2.3.2 Example

To demonstrate how SBFL can be applied listing 2.1 shows a buggy C function that can be used to sort rational numbers. On line 8, RationalGT is a function that compares whether the first fraction is greater than the second fraction. For SBFL is is not necessary to know how the function works internally as it can simply be treated as a black-box that is either executed or not executed. The four blocks that are used to locate the fault are marked with comments and the fourth block contains the actual fault as the swapping of the denominator is not implemented.

Listing 2.1. A buggy C function used to sort rational numbers. Taken from Abreu et al. [2006]

```
1 void RationalSort(int n, int *num, int *den) {
2     /* block 1 */
3     int i, j, temp;
4     for ( i = n-1; i>=0; i-- ) {
5         /* block 2 */
6         for ( j=0; j<i; j++ ) {
7             /* block 3 */
8             if (RationalGT(num[j], den[j], num[j+1], den[j+1])) {
```

```
9                   /* block 4; bug: forgot to swap denominator */
10                  temp = num[j];
11                  num[j] = num[j+1];
12                  num[j+1] = temp;
13              }
14          }
15      }
16  }
```

Table 2.1 shows a set of spectra for the RationalSort function containing all possible block hit spectra. Spectra S_1 to S_3 did not lead to a failure, as the faulty block 4 was never executed. Spectra S_4 did execute the faulty block, but is not flagged as erroneous (for example, if all denominators are equal it does not matter if they are swapped or not). This demonstrates that not every execution of a faulty statement leads to a failure in the output. As SBFL generally expects the faulty statements to have high n_{if} and low n_{ip} values, spectra S_4 introduces noise to the spectra. Spectra S_5 is the failing spectra that executed all blocks.

Table 2.1. Example set of spectra for the RationalSort function defined in listing 2.1.

	S_1	S_2	S_3	S_4	S_5	n_{ip}	n_{if}	n_{np}	n_{nf}	Tarantula	Ochiai
block 1	1	1	1	1	1	4	1	0	0	0.50	0.45
block 2	0	1	1	1	1	3	1	1	0	0.57	0.50
block 3	0	0	1	1	1	2	1	2	0	0.67	0.58
block 4	0	0	0	1	1	1	1	3	0	**0.80**	**0.71**
error	0	0	0	0	1						

The table also shows the $\langle n_{np}, n_{nf}, n_{ip}, n_{if} \rangle$ values for all blocks. For example, the faulty block 4 is involved in 1 passing execution, 1 failing execution and it is not involved in 3 passing executions. The suspiciousness scores for each block are calculated by the well-known Tarantula [Jones et al., 2002] and Ochiai Abreu et al. [2006] ranking metrics and shown in their respective columns. Block 4 is the most suspicious block considering both metrics and indeed contains the fault location.

2.3.3 Ranking Metrics

In this study the performance of multiple ranking metrics is compared and evaluated. All used ranking metrics are shown in figure 2.1 and are originally summarized by Naish et al. [2011].

Name	Formula	Name	Formula
Jaccard	$\frac{a_{ef}}{a_{ef}+a_{nf}+a_{ep}}$	Anderberg	$\frac{a_{ef}}{a_{ef}+2(a_{nf}+a_{ep})}$
Sørensen-Dice	$\frac{2a_{ef}}{2a_{ef}+a_{nf}+a_{ep}}$	Dice	$\frac{2a_{ef}}{a_{ef}+a_{nf}+a_{ep}}$
Kulczynski1	$\frac{a_{ef}}{a_{nf}+a_{ep}}$	Kulczynski2	$\frac{1}{2}\left(\frac{a_{ef}}{a_{ef}+a_{nf}}+\frac{a_{ef}}{a_{ef}+a_{ep}}\right)$
Russell and Rao	$\frac{a_{ef}}{a_{ef}+a_{nf}+a_{ep}+a_{np}}$	Hamann	$\frac{a_{ef}+a_{np}-a_{nf}-a_{ep}}{a_{ef}+a_{nf}+a_{ep}+a_{np}}$
Simple Matching	$\frac{a_{ef}+a_{np}}{a_{ef}+a_{nf}+a_{ep}+a_{np}}$	Sokal	$\frac{2(a_{ef}+a_{np})}{2(a_{ef}+a_{np})+a_{nf}+a_{ep}}$
M1	$\frac{a_{ef}+a_{np}}{a_{nf}+a_{ep}}$	M2	$\frac{a_{ef}}{a_{ef}+a_{np}+2(a_{nf}+a_{ep})}$
Rogers-Tanimoto	$\frac{a_{ef}+a_{np}}{a_{ef}+a_{np}+2(a_{nf}+a_{ep})}$	Goodman	$\frac{2a_{ef}-a_{nf}-a_{ep}}{2a_{ef}+a_{nf}+a_{ep}}$
Hamming etc.	$a_{ef}+a_{np}$	Euclid	$\sqrt{a_{ef}+a_{np}}$
Ochiai	$\frac{a_{ef}}{\sqrt{(a_{ef}+a_{nf})(a_{ef}+a_{ep})}}$	Overlap	$\frac{a_{ef}}{\min(a_{ef},a_{nf},a_{ep})}$
Tarantula	$\frac{\frac{a_{ef}}{a_{ef}+a_{nf}}}{\frac{a_{ef}}{a_{ef}+a_{nf}}+\frac{a_{ep}}{a_{ep}+a_{np}}}$	Zoltar	$\frac{a_{ef}}{a_{ef}+a_{nf}+a_{ep}+\frac{10000a_{nf}a_{ep}}{a_{ef}}}$
Ample	$\left\lvert\frac{a_{ef}}{a_{ef}+a_{nf}}-\frac{a_{ep}}{a_{ep}+a_{np}}\right\rvert$	Wong1	a_{ef}
Wong2	$a_{ef}-a_{ep}$		
Wong3	$a_{ef}-h,$ where $h=\begin{cases} a_{ep} & \text{if } a_{ep}\leq 2 \\ 2+0.1(a_{ep}-2) & \text{if } 2<a_{ep}\leq 10 \\ 2.8+0.001(a_{ep}-10) & \text{if } a_{ep}>10 \end{cases}$		
Ochiai2	$\frac{a_{ef}a_{np}}{\sqrt{(a_{ef}+a_{ep})(a_{np}+a_{nf})(a_{ef}+a_{nf})(a_{ep}+a_{np})}}$		
Geometric Mean	$\frac{a_{ef}a_{np}-a_{nf}a_{ep}}{\sqrt{(a_{ef}+a_{ep})(a_{np}+a_{nf})(a_{ef}+a_{nf})(a_{ep}+a_{np})}}$		
Harmonic Mean	$\frac{(a_{ef}a_{np}-a_{nf}a_{ep})((a_{ef}+a_{ep})(a_{np}+a_{nf})+(a_{ef}+a_{nf})(a_{ep}+a_{np}))}{(a_{ef}+a_{ep})(a_{np}+a_{nf})(a_{ef}+a_{nf})(a_{ep}+a_{np})}$		
Arithmetic Mean	$\frac{2a_{ef}a_{np}-2a_{nf}a_{ep}}{(a_{ef}+a_{ep})(a_{np}+a_{nf})+(a_{ef}+a_{nf})(a_{ep}+a_{np})}$		
Cohen	$\frac{2a_{ef}a_{np}-2a_{nf}a_{ep}}{(a_{ef}+a_{ep})(a_{np}+a_{ep})+(a_{ef}+a_{nf})(a_{nf}+a_{np})}$		
Scott	$\frac{4a_{ef}a_{np}-4a_{nf}a_{ep}-(a_{nf}-a_{ep})^2}{(2a_{ef}+a_{nf}+a_{ep})(2a_{np}+a_{nf}+a_{ep})}$		
Fleiss	$\frac{4a_{ef}a_{np}-4a_{nf}a_{ep}-(a_{nf}-a_{ep})^2}{(2a_{ef}+a_{nf}+a_{ep})+(2a_{np}+a_{nf}+a_{ep})}$		
Rogot1	$\frac{1}{2}\left(\frac{a_{ef}}{2a_{ef}+a_{nf}+a_{ep}}+\frac{a_{np}}{2a_{np}+a_{nf}+a_{ep}}\right)$		
Rogot2	$\frac{1}{4}\left(\frac{a_{ef}}{a_{ef}+a_{ep}}+\frac{a_{ef}}{a_{ef}+a_{nf}}+\frac{a_{np}}{a_{np}+a_{ep}}+\frac{a_{np}}{a_{np}+a_{nf}}\right)$		

Figure 2.1. The ranking metrics used for this study as presented by Naish et al. [2011]. Note that $a_{np}=n_{np}$, $a_{nf}=n_{nf}$, $a_{ep}=n_{ip}$, $a_{ef}=n_{if}$, as Naish et al. [2011] use different identifiers.

2.4 Evaluation of SBFL Research Methods

Parnin and Orso [2011] have studied how SBFL is used by developers in practice by conducting a human study. The study has shown that common assumptions by researchers are not met in practice. For example, it is commonly assumed that a developer can judge whether a program element contains a bug or not by just looking at it in isolation. However, the case study reveals that developers actually looked at the faulty statement, but inspected various other elements until returning to the faulty element. In addition to that the study noticed that developers do not linearly follow the ranked elements, but rather use the ranking to *search* for the fault location. The results of the study imply that most of the metrics used to measure the performance of SBFL are not valid and need to be improved.

For program versions in which multiple faults are present, DiGiuseppe and Jones [2014] have shown that multiple faults can interfere with each other. Fault location interference is defined as a decrease in fault localization performance that occurs due to the presence of specific faults. This implies that if multiple faults are present, some could have a bad fault localization performance simply due to the presence of another fault (e.g. one fault alters the execution flow such that the other fault is never executed). To overcome this limitation they propose to always find and fix the first bug in the ranking and then execute SBFL again.

Herzig et al. [2013] have examined bug databases of open-source projects and found that many filed bugs are wrongly classified. In fact, they report that 39% of bugs were not a bug, but rather a refactoring, enhancement or could be assigned to other categories that are not a bug. The misclassification implicitly introduces biased and wrong data into automatically mined bug databases and benchmarks and as such, those have to be carefully examined before used for empirical investigations.

Chapter 3

Research Methods

This chapter first presents the research questions behind this study. Next, commonly used performance metrics are introduced and defined. Lastly, the chapter presents new performance metrics defined by this study.

3.1 Research Questions

To understand how spectrum-based fault localization (SBFL) techniques can be applied to large real-world projects, 33 SBFL ranking metrics are applied to real-world faults. The following research questions guide the experiment design and the evaluation metrics:

Research Question 1. How does the program size influence fault localization performance?

The research community uses the Software-artifact Infrastructure Repository (SIR) [Do et al., 2005] as benchmark suite for fault localization techniques. However, most real-world software is larger than the software currently present in the SIR. Therefore this research questions analyzes whether the results of the current research can be applied to bigger real-world projects.

Research Question 2. How many bugs can be found when examining a fixed amount of ranked elements?

Parnin and Orso [2011] have shown that developers do not have a significant benefit by debugging with SBFL in practice. They also found that developers quickly loose trust to the SBFL tool if the first ranked elements do not point them to relevant program elements. The motivation behind this research question is to examine how the fault locations in the first ranked elements behave for large software systems.

Research Question 3. How does the program size influence the suspiciousness scores produced by different ranking metrics?

SBFL techniques assign a suspiciousness score to all program elements. To enable a meaningful interpretation of the results of a SBFL technique it is important to understand how the program size can influence the assigned suspiciousness scores.

3.2 Definitions

This section introduces common definitions required to define performance metrics for SBFL techniques. A SBFL technique R is a function that assigns a suspiciousness score to each program element $c_i \in C$, which is denoted as $susp_R(c_i)$. C is the set of all components such that $\forall c_j, j \in \mathbb{N}^+ : c_j \in C$ holds. Furthermore, the set of faulty statements is denoted as $F \subseteq C$.

Prominent Bug

Faulty programs may contain multiple bugs. DiGiuseppe and Jones [2014] have shown that multiple faults interfere with each other such that some faults can not be localized using SBFL if others are present at the same time. To overcome this limitation they propose to iteratively run SBFL, locate the first fault, fix the fault and then start over again until all bugs are fixed. With this approach the first bug that is found is the most important bug and they denote it as *prominent bug*. The prominent bug $f_{pro} \in F$ is then defined as follows:

$$\forall f_j \in F \backslash \{f_{pro}\} : susp_R(f_{pro}) \geqslant susp_R(f_j) \tag{3.1}$$

Rank

Performance of SBFL algorithms is usually evaluated by examining the ranking position of the faulty element in the final ranking. However, the ranking position may not be deterministic, as it can occur that multiple ranked elements share the exact same suspiciousness. As SBFL has no additional information on how to rank draws, all elements with the same suspiciousness are randomly ordered. If there are multiple elements with the same suspiciousness as the faulty element, the final ranking thus has a best case and a worst case. In the best case the faulty element is the first element of all elements with the same suspiciousness:

$$best\_rank_R(c_j) = \left|\{c_i \,|\, sus_R(c_i) > sus_R(c_j)\}\right| \tag{3.2}$$

In the worst case the faulty element is the last element of all elements with the same suspiciousness:

$$worst\_rank_R(c_j) = \left|\{c_i \,|\, sus_R(c_i) \geqslant sus_R(c_j)\}\right| \tag{3.3}$$

As the random order follows a uniform distribution the average case is defined by:

$$average\_rank_R(c_j) = \frac{1}{2}\left(best\_rank_R(c_j) + worst\_rank_R(c_j)\right) \tag{3.4}$$

With these ranking functions it is possible to create performance metrics for SBFL techniques as defined in the following sections.

3.3 Common Performance Metrics for SBFL

3.3.1 Wasted Effort (WE)

Current research practice assumes that developers have a perfect fault understanding and can identify the fault as soon as they reach the fault location while traversing the ranking list. With this assumption, every developer has to inspect all the elements that are ranked higher than the faulty element. The number of inspected non-faulty elements is thus defined as the wasted effort for the developer. Current research commonly puts the wasted effort in relation to the total number of ranked elements and as such the wasted effort is a percentage defined as:

$$min\_we_R(c_j) = \frac{best\_rank_R(c_j) - 1}{|C|} \tag{3.5}$$

$$max\_we_R(c_j) = \frac{worst\_rank_R(c_j) - 1}{|C|} \tag{3.6}$$

As this metric is defined for faulty spectra elements and not the ranking metric itself, the results of this metric need to be aggregated by an aggregation function a, for example by taking the average, to compute a score for a specific ranking metric:

$$min\_we\_aggregated(R) = a(\{we \mid we \in min\_we_R(c_j) \wedge c_j \in C\}) \tag{3.7}$$

$$max\_we\_aggregated(R) = a(\{we \mid we \in max\_we_R(c_j) \wedge c_j \in C\}) \tag{3.8}$$

Using the aggregated score, different ranking metrics can be compared against each other.

3.3.2 Proportion of Bugs Localized (PBL)

Another commonly used metric is the proportion of bugs localized when examining a certain percentage of code. This metric is defined for a ranking metric and directly produces a score that can be compared to other ranking metrics:

$$min\_pbl_p(R) = \frac{|\{f \mid f \in F \wedge min\_we_R(f) \leqslant p\}|}{|F|} \tag{3.9}$$

$$max\_pbl_p(R) = \frac{|\{f \mid f \in F \wedge max\_we_R(f) \leqslant p\}|}{|F|} \tag{3.10}$$

where $p \in [0, 1]$ is the percentage of code inspected.

3.3.3 Hit@X

Lucia et al. [2014] have introduced a new fault localization performance metric called "Hit@10". The key idea is to count how many bugs can be found by investigating a fixed

amount of ranked elements. The metric addresses the issues Parnin and Orso [2011] have raised, as developers only investigate an absolute number of ranked elements before giving up and using alternate debugging methods. The authors have chosen 10 basic blocks as threshold for the metric, but for this study the metric is generalized to have a varying threshold of X ranked elements. The metric is defined as follows:

$$min\_hit\_count_X(R) = |\{f \mid f \in F \wedge best\_rank_R(f) \leqslant X\}| \tag{3.11}$$

$$max\_hit\_count_X(R) = |\{f \mid f \in F \wedge worst\_rank_R(f) \leqslant X\}| \tag{3.12}$$

$X \in \mathbb{N}^+$ represents the number of elements inspected. To be able to better compare the metric, it is possible to define the metric relative to the number of faults available:

$$min\_hit_X(R) = \frac{min\_hit\_count_X(R)}{|F|} \tag{3.13}$$

$$max\_hit_X(R) = \frac{max\_hit\_count_X(R)}{|F|} \tag{3.14}$$

3.3.4 Suspiciousness

DiGiuseppe and Jones [2014] compare the suspiciousness of a faulty program element to the mean suspiciousness of all non-faulty program elements. If the suspiciousness of a faulty program element is below or near the mean suspiciousness the faulty program element does not stand out from the body of non-faulty elements and is then difficult to detect for a developer. The mean suspiciousness is the arithmetic average of all suspiciousness scores of non-faulty elements:

$$mean\_susp(R) = \frac{1}{|C \backslash F|} \cdot \sum_{c_j \in C \backslash F} susp_R(c_j) \tag{3.15}$$

Ranking metrics can assign suspiciousness scores of any real number. As applied by Lucia et al. [2014], to compare these scores between ranking metrics all scores are normalized to fit into the interval $[0, 1]$. To compute the normalized score the minimum and maximum suspiciousness assigned by a ranking metric for a set of program elements C is required:

$$susp\_min_R(C) = susp_R(c_j), \forall i, i \neq j, 1 \leqslant i \leqslant |C| : susp_R(c_j) \leqslant susp_R(c_i) \tag{3.16}$$

$$susp\_max_R(C) = susp_R(c_j), \forall i, i \neq j, 1 \leqslant i \leqslant |C| : susp_R(c_j) \geqslant susp_R(c_i) \tag{3.17}$$

The normalized suspiciousness can then be computed by:

$$susp\_norm_R(c_j) = \frac{susp_R(c_j) - susp\_min_R(C)}{susp\_max_R(C) - susp\_min_R(C)} \tag{3.18}$$

3.4 New SBFL Performance Metrics

The new performance metrics use the Proportion of Bugs Localized (PBL) and the Hit@X metric to create discrete cumulative functions that can be analyzed. The PBL metric and the Hit@X metric both have a varying parameter that can be used to turn the metric into a discrete cumulative function consisting of a set of equidistant points $(x_i, y_i), i \in [0, n], n \in \mathbb{N}^+$. For a fixed number of support points n the set of support points for the PBL metric is defined as

$$x_i = \frac{i}{n} \qquad y_i = (min|max)\_pbl_{x_i}(R)$$

and for the Hit@X metric as

$$x_i = \frac{i}{n} \cdot |F| \qquad y_i = (min|max)\_hit_{x_i}(R).$$

The trapezoidal rule is leveraged to calculate the area under the curve of a specific metric M:

$$AUC(M) = \frac{1}{n} \cdot \left(\frac{y_0}{2} + \sum_{i=1}^{n-1} y_i + \frac{y_n}{2} \right) \tag{3.19}$$

3.4.1 Area Between Curves (ABC)

The past metrics definitions always define a metric for the best case and the worst case (and implicit due to the uniform distribution of the ranking also an average case). However, if the best case and worst case of any metric diverge there have to be ranked elements with the same suspiciousness in the ranking implying a random order to parts of the ranking. From a practical point of view, a random ranking does not make sense. If there is lots of randomness in the ranking, a developer seeking guidance in the debugging process must have a good portion of luck to find bugs, as the quality of the ranking is not reliable. The more the best and average case diverge, the more randomness is present in the ranking and as such it is less reliable. To measure the randomness of a given best/worst case metric the area between the best and worst-case curves can be leveraged. As $M_{worst} \leqslant M_{best}$ holds in all points per definition, the area between the curves is computed by:

$$ABC(M) = AUC(M_{best}) - AUC(M_{worst}) \tag{3.20}$$

The ABC metric reflects the decidedness of a ranking metric. If the randomness in the ranking is low, ABC is close to zero and the decidedness of the ranking metric is high. On the other hand, if the randomness in the ranking is high, ABC is high and the decidedness of the ranking metric is low. If not otherwise noted, throughout this study the ABC value is presented as percentage of the maximum area that can be achieved by a completly diverging best and worst case. This makes the metric easily comparable if the x-axis or y-axis scale is different.

3.4.2 Weighted Area Under Curve (WAUC)

For the discrete cumulative functions of the Proportion of Localized Bugs (PBL) and the Hit@X metric it is important that the curves rise as quickly as possible. The faster the curve rises, the less effort a developer has while finding even more bugs. To create a measure for the fast rising in the beginning, a weighting function $W(x)$ is applied to the cumulative curve of the performance metric M. The weighting function is defined on the interval $[0, 1]$ where the left end represents no effort for the developer and the right end represents the maximum effort for a developer. The WAUC value is then defined as:

$$WAUC(M) = AUC(M_S(x) \cdot W(x)) \tag{3.21}$$

where $M_S(x)$ is a performance metric that has been scaled to fit in the interval $[0, 1]$ and $x \in [0, 1]$ represents the effort a developer has applied. W then has to fulfill the following criteria:

1. W has to be an exponential function.
2. $\int_0^1 W(x)dx = 1$. A perfect debugging result finds all bugs with no effort, such that $M(x) = 1$. If the integral of the weighting function is 1, the resulting WAUC value is exactly 1. On the other hand, the worst debugging result is represented by $M(x) = 0$ where no bugs are found regardless of the effort. In such a case, the WAUC value is exactly 0. Any given metric with $M(x) \in [0, 1]$ will then yield a WAUC value in the interval $[0, 1]$. WAUC values can then be easily compared to each other.
3. $W(0.5) < 0.2$. After a developer wastes more than half of the maximum possible effort, the weight for the remaining curve has to be lower than 0.2.

The following function has been constructed from $f_{a,b}(x) = a \cdot e^{-b \cdot x}$ in order to fulfill criteria 1 and 2:

$$W_b(x) = \frac{b \cdot e^b}{e^b - 1} \cdot e^{-b \cdot x} \tag{3.22}$$

Proof for criteria 2:

$$\begin{aligned} \int_0^1 W_b(x)dx &= \left[-\frac{e^b}{e^b - 1} \cdot e^{-b \cdot x} \right]_0^1 \\ &= -\frac{e^b}{e^b - 1} \cdot e^{-b} + \frac{e^b}{e^b - 1} \\ &= \frac{e^b \cdot (-e^{-b} + 1)}{e^b - 1} \\ &= \frac{-e^0 + e^b}{e^b - 1} \\ &= 1 \qquad \blacksquare \end{aligned} \tag{3.23}$$

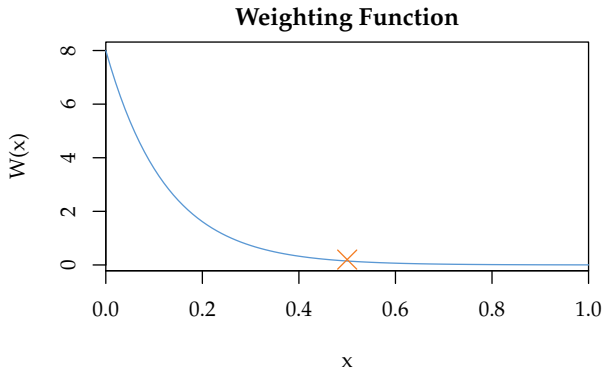


Figure 3.1. The used weighting function W_8. Criteria 3 is marked by the cross.

To fulfill the third criteria, $b = 8$ is chosen. The resulting weighting function is shown in Figure 3.1. Throughout this study the WAUC value is denoted as a percentage, for example a WAUC of 0.2345 is denoted as 23.45%.

3.4.3 Number of Files Investigated (NFI)

Parnin and Orso [2011] have shown in their study that developers do not linearly follow the ranking produced by SBFL. Instead, they use the statements ranked high by SBFL as starting points for their investigation and then *search* for the actual fault location. This observation indicates that it is more important to point developers to good starting points using SBFL than to improve the ranking of the fault locations itself. When visiting a ranked line proposed by SBFL, a developer has three options: he may either quit searching for the fault in the surrounding area, he may investigate the method, class, or file the ranked statement belongs to, or he may continue to search in other places by investigating the methods or classes *used* by the current surrounding area. He may also choose to wisely combine the three options.

To assess the performance of pointing a developer to the right places using SBFL, the following assumptions concerning the developers fault localization behavior are made:

- The developer is able to recognize the fault location as soon as he sees the fault location. He can then immediately stop localizing the fault.
- The developer will follow the list of ranked statements in a linear order.
- The developer will only visit the ranked element, if he has not visited the file containing the ranked element before.
- The developer will always investigate the whole file to which the currently visited element belongs to.

Generally speaking, with the given assumptions the *number of files* a developer has to investigate when using the lines in the SBFL ranking as starting points, is determined. The metric can be defined for the best- and the worst-case as follows, where $file(c_k)$ returns the file name of the program element c_k:

$$min\_nfi_R(c_j) = \left|\{file(c_k) \mid best\_rank_R(c_k) \leqslant best\_rank_R(c_j)\}\right| \tag{3.24}$$

$$max\_nfi_R(c_j) = \left|\{file(c_k) \mid worst\_rank_R(c_k) \leqslant worst\_rank_R(c_j)\}\right| \tag{3.25}$$

The assumptions clearly do not model the case that the developer may decide to continue the investigation in methods or classes used by the current surrounding area. Nevertheless, the assumptions allow to evaluate whether it is *feasible* for a developer to embrace such a fault localization process or not.

Chapter 4

Case Study: AspectJ

Throughout the research community SBFL has mostly been applied to and evaluated against projects of smaller size or to projects from the SIR. To understand the impact of large real-world software systems on SBFL performance AspectJ is used as case study for this work. AspectJ has been used by Dallmeier and Zimmermann [2007] to extract a benchmark suite of real-world bugs that were semi-automatically mined from the project history and thus provides an ideal foundation for this study.

This chapter first presents background information on the AspectJ project. The chapter then describes project characteristics of the AspectJ project that can be used to compare the AspectJ project to other existing projects. Next, the experimental setup and the bug classification process is described. Finally, the chapter describes the metrics used to evaluate the fault localization performance.

4.1 AspectJ Project Background

AspectJ [Asp] is an extension to the Java programming language that enables developers to use an aspect-oriented programming style. The aspects are transformed to native Java code to retain cross-platform compatibility, but support developers to easily implement crosscutting concerns, such as monitoring or error handling that affect a large part of the codebase of a software system. AspectJ was released to the open-source community in 2001 and has since been actively developed.

Due to the size and the long available development history of the project Dallmeier and Zimmermann [2007] have used AspectJ to introduce iBugs. iBugs is an approach that automatically mines a software repository with an associated issue tracker to extract bugs with their associated code fixes. The mining basically matches keywords such as *fix* or *bug* in the commit messages and associates them with issues found in the issue tracker. The mined set of bugs was then manually inspected to remove false positives. In addition to that, only bug fixes that actually change the source code, have compiling pre-fix and post-fix versions and have compiling test suites were kept in the set of bugs. From an initial set of 489 bug candidates, the research team reduced the AspectJ dataset to 369 bugs after applying the aforementioned restrictions. Given the guarantee that the pre-fix version of each bug compiles and with build and test execution ant scripts the research team provided, the iBugs benchmark suite suits perfectly as case study for this work.

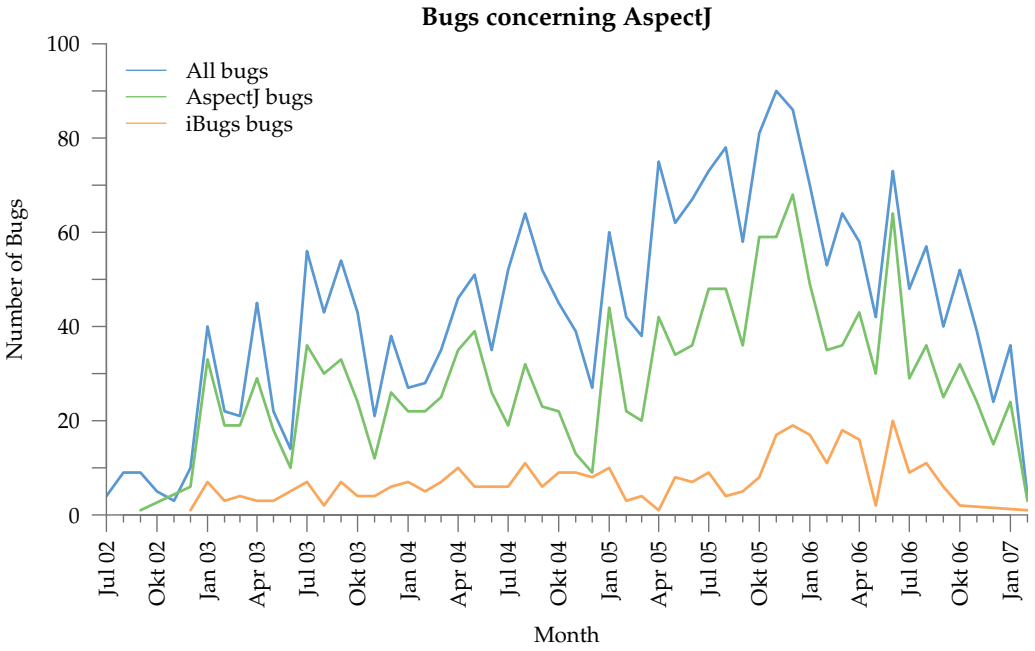


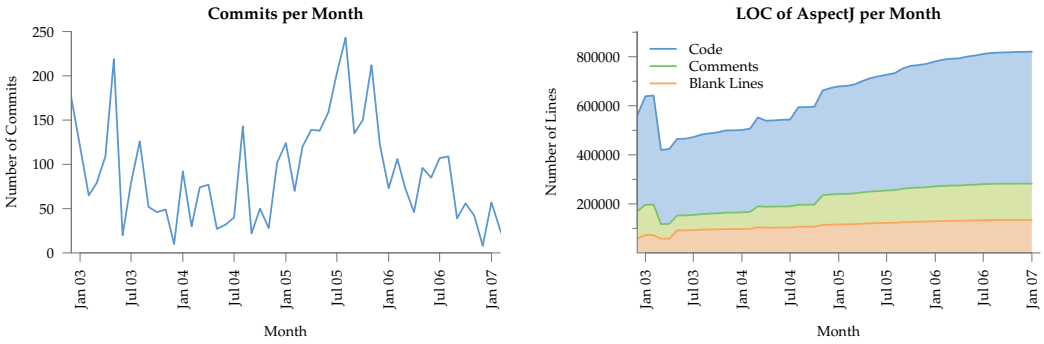
Figure 4.1. Displays the number of bugs for the AspectJ project within the time frame that was examined in this study

4.2 AspectJ Project Characteristics

The 369 bugs in the AspectJ iBugs benchmark suite range from the AspectJ bug id 28919 to 173602 and were reported between December 30, 2002 and February 9, 2007, spanning four years of development history. Each bug is either assigned to the product AJDT or AspectJ. AJDT are the AspectJ development tools which provide tool support for AspectJ development for the Eclipse platform. The AJDT bugs affect the components *Core* and *UI*. The AspectJ bugs affect the components *AJBrowser*, *AJDoc*, *Ant*, *Compiler*, *IDE*, *Library* and *LTWeaving*.

Line "All bugs" in Figure 4.1 (data taken from Openhub [2014a]) plots the number of bugs filed per month for all aforementioned components. The line "AspectJ bugs" shows all bugs that are assigned to the product AspectJ (thus not AJDT) and the line "iBugs bugs" shows the bugs that are part of the iBugs benchmark suite. In most month there were 20-90 bugs assigned in total. The iBugs benchmark suite consistently contains bugs of nearly every month in the project history. The suite is thus representative for the bugs that occured in the whole development time span.

In the examined development timespan the contributors have produced a total of 7677 commits. Figure 4.2a (data taken from Openhub [2014b]) shows the commits per month



(a) AspectJ commits per month

(b) AspectJ lines of code per month

Figure 4.2. Key metrics of the AspectJ project history. The faulty versions for this study were taken from the same time frame as displayed in the plots.

during the development timespan. The project has been actively developed in the examined time frame.

Figure 4.2b (data taken from Openhub [2014b]) shows the evolvement of the lines of code within the development time frame. After the initial release there was a fairly large drop in the lines of code, but the project then steadily grew in size. Note that the presented lines of code also include test sources and non-Java source files.

4.3 Experimental Setup and SBFL Limitations

AspectJ has been prepared by Dallmeier and Zimmermann [2007] for the iBugs benchmark suite. The research team provides a set of Ant scripts with iBugs that allows to extract the source files of a specific bug, compile the sources and test sources and then execute the unit tests of AspectJ. As it is possible to execute each single unit test by itself, iBugs provides a framework to easily generate the necessary data for SBFL. Cobertura[1] was used to instrument the AspectJ sources and generate the coverage reports as XML file. The whole process of generating the coverage files for a specific bug was automated using custom scripts.

4.3.1 Data Preparation

Dallmeier and Zimmermann [2007] have included the actual fixes that were applied to fix the bug for every buggy version. This was done by extracting the change set that was checked into the version control system by the AspectJ developer who fixed the bug. In most cases however, the change set included a lot of changed lines and files that did not

[1]Cobertura project website: http://cobertura.github.io/cobertura/

fix the actual bug. To evaluate the quality of the rankings produced by the various SBFL algorithms in this study, it is of high importance that the change set consists of only the lines that really contained the fault. To obtain a smaller change set of only the actual fixes, all lines of the change sets used in this study were manually classified. It is assumed that the given change set from the iBugs benchmark suite includes the real fault and then all lines in the change set were either classified as *buggy* or *healthy*. Next to the classification of the line itself a level of confidence to which the classification is correct was stored. The confidence was distinguished between low, normal and high confidence and additionally contains an explanatory comment containing the reasoning for all the lines classified as *buggy*.

In total, 110 buggy versions were classified and only 42 were found to be applicable for SBFL. The 68 removed buggy versions were not applicable for SBFL for the following reasons: 25 buggy versions were classified as *enhancement* and not as a bug. Section 4.3.3 will expound upon the difference between bugs and enhancements. An additional 36 bugs did not contain a single line in the changeset that was classified as fault location, because some of the change sets did not include changes to Java source code at all while others did change Java source code that does not appear in coverage reports (e.g. refactoring names, changing imports, ...). The remaining 7 bugs either had only faulty lines classified with low confidence or did not produce any execution traces due to compile time or runtime errors. From the set of 42 buggy versions, only fault locations of 21 versions were executed by at least one test case.

Table 4.1 shows the number of bugs and how the set of bugs was reduced through the various stages of data preparation. Nearly 30% of the bugs contained in the iBugs benchmark suite were classified for this study, from which less than 40% were applicable for SBFL and less than 20% were actually involved in at least one test case. Future research should improve the data preparation process as the time required to yield empirical results of good quality is too large to be feasible for researchers. In addition to that, researchers should be encouraged to publish their data sets in a unified format.

4.3.2 Data Quality Impacts

While producing the coverage files for the various buggy version of AspectJ several issues that impact the quality of the data were noticed.

First of all, not all provided tests of AspectJ were executable using our setup. Judging from the error messages it seemed that the errors occur due to platform incompatibilities. As the suggested versions (Java 1.4.2, Ant 1.6.5) by Dallmeier and Zimmermann [2007] of the tools to use were released between 2005 and 2006 it was difficult to reproduce the execution environment. The errors occurred mostly during test execution and resulted in empty coverage traces (not a single line was hit). In such a case the produced coverage trace was removed from the dataset.

Secondly, not all test cases were executed, as executing and instrumenting each single test case was not possible within the time scope of this work. The executed test cases were

Table 4.1. The number of bugs associated to the different components. Provides an overview on how many bugs exist, how many have been prepared for the iBugs benchmark suite and how many have been further examined for this study.

Product	**AspectJ**							**AJDT**		**Sum**
Component	AJBrowser	AJDoc	Ant	Compiler	IDE	Library	LTWeaving	Core	UI	
All AspectJ bugs	9	21	33	1287	106	10	78	409	477	2430
Bugs in iBugs	1	8	4	231	19	3	19	19	46	350
Classified bugs	0	1	1	77	9	0	11	4	7	110
Applicable bugs	0	0	0	33	5	0	3	1	0	42
Involved bugs	0	0	0	17	2	0	1	1	0	21

chosen randomly. However, all *failing* test cases were executed, as the number of failing test cases is between 24 and 191 for all buggy versions. In addition to that, at least as many *passing* test cases as failing test cases were produced.

Lastly, it occurred that the actual fault location was often not executed by a single failing or passing test. If a fault location was never involved in a single failing test case for a buggy version there are two explanations for this:

- The fault was covered by a test case that failed to execute with our setup. Obviously, if no test case failed this argument does not hold.

- The fault location is not covered by a single failing test case and all randomly chosen passing test cases of the AspectJ test suite.

4.3.3 Fault Classification

Dallmeier and Zimmermann [2007] provide the change set between pre- and post-fix versions in the diff[2] format between the two corresponding revisions in the version control system. As the changed lines the developers commit mostly exceed only the real fault location, the change sets had to be manually classified in order to be able to use the data as benchmark for fault localization. During the classification several bugs which are difficult to localize using SBFL were encountered. This section summarizes the anecdotal evidence found.

[2]http://www.gnu.org/software/diffutils/

Misclassification of Bug Reports

25 of 110 of all manually classified versions were not a real bug. This was reasoned from both the bug description and the change set itself. The following rules are a subset of the rules presented by Herzig et al. [2013] which are used to distinguish bugs from non-bugs and are tailored towards the bugs encountered in AspectJ. A bug report *increases* the likelihood of the bug report being a real bug, if

- it reports a NullpointerException (most common exception type) or other types of exceptions.
- a minimal failing example was provided that lead to small semantic code changes fixing the issue.
- it reports runtime issues caused by defects (e.g. endless loops).
- ...

In addition to that there are also rules that *lessen* the likelihood of a bug report being a real bug, namely if

- the phrases "it would be great if", "improve" or "enhance" or similar phrases appeared in the bug description. No automatic classification was performed, but the presence of these words have a high correlation with the bug report not being a real bug, but rather an enhancement.
- the report contains request to add new features.
- the change set only contained refactored code.
- only strings (exception strings, log strings, ...) were changed in the change set.
- there are changes in the change set that were made to help debugging (and have been committed).
- ...

There are cases where rules that lessen and rules that increase the likelihood can be applied. The classification was performed in a restrictive manner, such that only bugs with a high likelihood of being a bug were actually classified as bug. The goal was to minimize the error of the first kind for the classification.

Bugs that are Difficult to Detect Using SBFL

There are some bugs that are difficult to detect with spectrum-based fault localization, because they are difficult to reproduce. Several bugs occurred due to concurrency issues or environment issues. Environment issues include for example hardware constraints that lead to `OutOfMemoryError`'s that were resolved by adding appropriate try-catch clauses. Another fix resolved a concurrency issue by adding a `synchronized` modifier to a method. Both example cases may be difficult to reproduce in a test suite within reasonable budget constraints and are thus difficult to detect using SBFL as they do not lead to failing tests in the suite.

Classification Summary

All in all, there were a lot of bugs that could be classified with high confidence without domain knowledge, but in edge cases it is error prone to find the real fault location without domain knowledge. Especially, a source of error furthermore is relying on the change set in the version control system to contain the real fault location, but without domain knowledge it is difficult to determine whether a developer has fixed the real fault or implemented a workaround. To ensure the data for this study is of high quality the bugs were classified in a restrictive manner and were rather excluded in case of inconclusiveness.

Chapter 5

Evaluation of Experimental Results

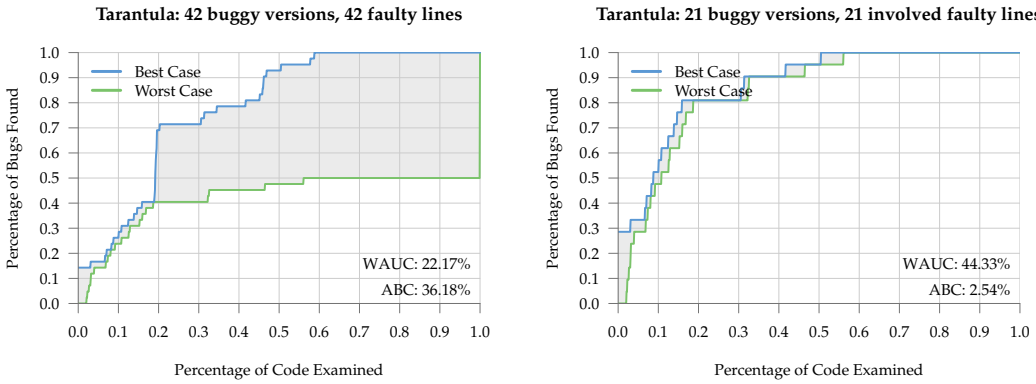
Based on the coverage data produced with the AspectJ iBugs benchmark suite this chapter evaluates why it is difficult to apply spectrum-based fault localization to real-world projects by answering the research questions defined in chapter 3.

5.1 Research Question 1: How does the program size influence fault localization performance?

5.1.1 Multiple Lines are Mapped to the Same Suspiciousness

Spectrum-based fault localization algorithms use simple arithmetics to compute the suspiciousness. The input for each fault localization algorithm are the four numbers $\langle n_{np}, n_{nf}, n_{ip}, n_{if} \rangle$: The number of passing executions not involving the statement, the number of failing executions not involving the statement, the number of passing executions involving the statement and the number of failing executions involving the statement. The sum of these four numbers is equal to the number of available execution traces. Hence, if less execution traces are available, the number of possible inputs to SBFL algorithms drastically shrinks. Due to the arithmetics that calculate the suspiciousness score using these four numbers, the amount of different suspiciousness scores is limited. In addition to that, different inputs can result in the same suspiciousness score, as, for example, the addition and the multiplication are associative operations that are not influenced by the order of the operators. For example the Hamming ranking metric $n_{if} + n_{np}$ as defined in Section 2.3.3 yields the same result for $n_{np} = 4, n_{if} = 6$ and $n_{np} = 6, n_{if} = 4$.

With an increasing number of program elements that need to be ranked it becomes increasingly important to assign a unique suspiciousness score to every single program element. The real drawback of multiple program elements that share the same suspiciousness arises when ranking the suspicious program elements in the order in which a developer should inspect them. If multiple program elements share the same suspiciousness, those elements do in fact have the same ranking position and as no further clue is available using plain SBFL the actual order of those elements can only be randomly chosen. However, this is actually the value SBFL intends to provide to developers in order to assist them during debugging: A structured walk-through through suspicious program elements. This randomness does not influence the developers investigation if only a few program elements



(a) Fault localization quality of all available bugs

(b) Fault localization quality of all bugs, that were involved in at least one execution trace

Figure 5.1. Tarantula fault localization decidedness

share the same suspiciousness, but the approach becomes increasingly worthless if more than hundreds to thousands of program elements share the same suspiciousness.

The examined versions of AspectJ contain between 15.788 and 210.458 lines of executable and coverage-producing code. For each version between 20 and 163 execution traces were produced. Under the given circumstances most suspiciousness scores in the ranking are shared by multiple lines. As the order of multiple lines with the same suspiciousness can only be randomly chosen, the Area Between the two Curves ("ABC") metric in section 3.4.1 has been defined to have a repeatable and meaningful metric for the randomness. In all plots the area used to calculate the ABC metric between the two curves is highlighted. The smaller the ABC metric, the less random a ranking is and as such, the results of the fault localization algorithm are more reliable for a developer debugging a program. A value of 100% represents a completely random ranking (low decidedness) whereas a value close to 0% (high decidedness) represents a deterministic ranking.

ABC for involved and non-involved faulty lines

Figure 5.1 shows the fault localization quality of the well-known Tarantula fault localizer [Jones et al., 2002]. Both figures each plot the best- and worst-case proportion of prominent bugs localized. Figure 5.1a includes all prominent bugs available in the dataset, whereas Figure 5.1b only includes all prominent bugs that were involved in at least one test case. In both plots, the horizontal axis represents the percentage of code examined and the vertical axis represents the percentage of bugs localized. If the best- and the-worst case are identical, only the best-case curve is visible. The curves reveal the minimum (worst-case) percentage of bugs a developer can find and the maximum (best-case) percentage of bugs

a developer can find when examining a certain percentage of the ranked lines. The two curves are discrete curves, but to visually see the trends of the curves better, the points of the curves are linearly connected in the plot.

In Figure 5.1 Tarantula maps a lot of fault locations to the same suspiciousness resulting in an ABC of 36.18%. After examining $\sim$ 50% of the code, the Tarantula ranking shows a very large divergence between the best- and worst-case ranking and thus becomes worthless from a practical point of view, as all remaining bugs are in random order. However, approximately half of the evaluated bugs were never involved in either a passing or failing execution trace. Under such a condition it is difficult for spectrum-based fault localization to locate the fault, as the fault location then shares the same suspiciousness with at least all the statements that also were never involved in a single execution trace.

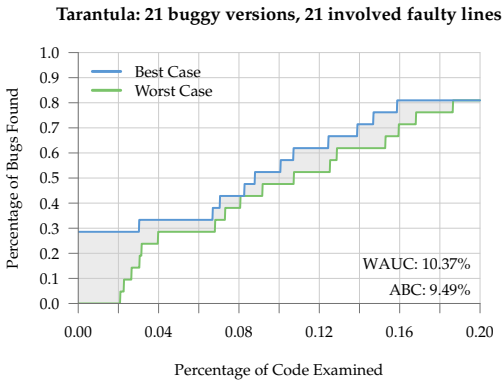
As shown in Figure 5.1b when only locating faults that are involved in at least one execution trace the fault localization performance greatly improves. Instead of locating only $\sim$ 25% of all faults after examining 10% of the code, if only fault locations that are involved are localized Tarantula is able to locate $\sim$ 50% of the faults after examining 10% of the code. In addition to that, the decidedness of Tarantula increases to an ABC of 2.54%, as the faults that share the same suspiciousness with the majority of the statements due to their non-involvement are not contained in the plot. However, it is important to keep in mind that 50% of all available bugs were removed to obtain the subset of bugs that were involved.

The data supports that program size can have a negative effect on fault localization performance when measured by the PBL metric by introducing randomness in the ranking. Nevertheless, program size is not the only factor responsible for introducing the randomness. The quality of the test suite and the quantity of test cases may also significantly impact the randomness in the ranking.

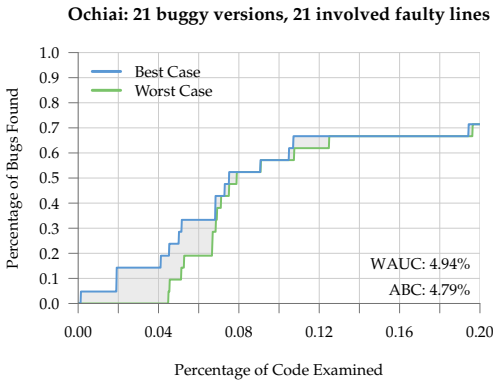
Impact of Different Metrics for Same Suspiciousness

In figure 5.1b the parts of the curve that contribute most to the ABC value are prior to 20% of examined code. However, spectrum based fault localization intends to point the developer to the faulty location with the first elements in the ranking and thus, algorithms should rank elements in the first positions with high confidence and decidedness. Examining the growth of the best- and worst-case rankings for the first 20% of examined code for the Tarantula fault localizer in figure 5.2a, it is evident that the guidance for a developer is not very clear. Depending on the random ranking a developer may find up to $\sim$ 30% of the bugs after examining $\sim$ 2% of the code or no bug at all. At this point it is especially important to note that 2% of 200.000 lines of code are 4.000 lines of code the developer has to inspect in the worst case until finding the *first* bug.

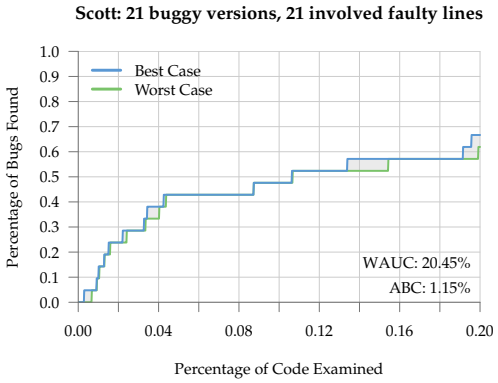
In figure 5.2b the ranking curves for the first 20% of examined code are shown for the Ochiai [Abreu et al., 2006] fault localizer. Compared to Tarantula, the decidedness of the Ochiai coefficient is definitely higher (ABC lower, Tarantula 9.49% vs. Ochiai 4.79%), though with Ochiai a maximum of 15% of the bugs within the first $\sim$ 2% of examined code



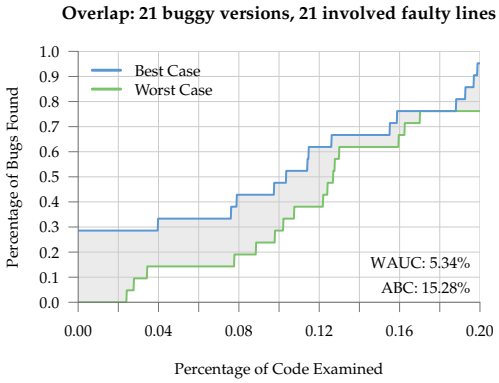
(a) A closer look at the growth of the ranking curves of Tarantula when examining the first 20% of the code.



(b) A closer look at the growth of the ranking curves of Ochiai when examining the first 20% of the code.



(c) A closer look at the growth of the ranking curves of Scott when examining the first 20% of the code.



(d) A closer look at the growth of the ranking curves of Overlap when examining the first 20% of the code.

Figure 5.2. Tarantula and Ochiai fault localization decidedness in the first 20% of examined code

can be found. Depending on the random ranking between 2% and 4% of examined code it is even possible to not find a single bug until $\sim$ 4% of examined code. Between 4% and 10% of examined code, the Ochiai metric rises very steep and after 10% of examined code the Ochiai metric has localized nearly $\sim$ 60% of the bugs that were involved at least once. Between 12% and 20% of examined code, the Ochiai metric does not find any bugs at all as the curves do not show any steps. Tarantula does not find any bugs between 4% and 6% of examined code, but then starts to steadily rise to find $\sim$ 80% of all bugs within the first

20% of examined code.

Contrary to common results of other researchers using the Software-artifact Infrastructure Repository [Do et al., 2005] as benchmark it cannot be confirmed that the Ochiai ranking metric outperforms the Tarantula ranking metric. In fact, Tarantula performs more than two times better in the first 4% of the inspected code in the average case, where Tarantula guarantees to find nearly 30% of all fault locations and Ochiai cannot guarantee to find even a single one.

Figure 5.2c and Figure 5.2d plot the ranking metrics with the lowest and the highest ABC value of all examined ranking metrics, respectively. Scott has the lowest ABC value of 1.15% within the first 20% of faults located and Overlap has the highest ABC value of 15.28% within the first 20% of faults located. Although Scott is a sophisticated ranking metric and Overlap a simple ranking metric in terms of arithmetic operations that need to be evaluated in order to compute the suspiciousness of a ranked element, the data contradicts a correlation between the number of arithmetic operations and the ABC value of a ranking. For example, the Wong1 ranking metric (not plotted) is equal to the number of failing executions the ranked element is involved in (n_{if}) and has an ABC value of 5.38%.

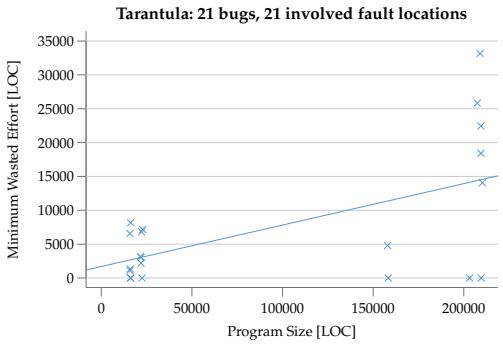
All in all, if multiple elements share the same suspiciousness spectrum-based fault localization reaches its limitations. The examined ranking metrics considerably differ in their ABC value for programs of large size and thus, the ABC value may be used as an indicator whether a given ranking metric is suited for fault localization for a given problem domain with a certain size. Future research may investigate how and why the program size and the size of the test suite influence the ABC value to better understand how the differences of ranking metrics can be leveraged to improve SBFL for real-world scenarios.

Observation 1. Large programs can introduce randomness in the ranking through a lot of ranked elements having the same suspiciousness. If the fault locations are guaranteed to be involved in at least one execution trace ranking metrics are much more decided resulting in lower ABC values. Different ranking metrics may have a major impact on the ABC value, but no correlation between the ranking metric and the ABC value was found.

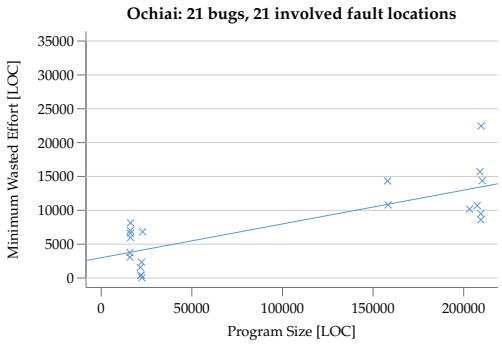
5.1.2 Wasted Effort

Figure 5.3a-Figure 5.3d display the wasted effort in lines of code (LOC) when localizing a fault in a buggy version of a specific size. The horizontal axis represents the size of the buggy version that was inspected and the vertical axis represents the minimum lines of code that need to be examined in order to find the prominent bug. The visible line has been linearly fitted to illustrate the trend of the data points.

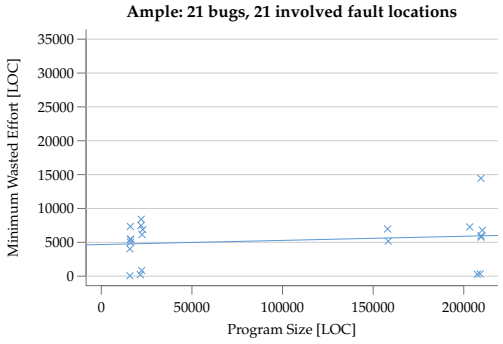
The plot for the Tarantula ranking metric is shown in Figure 5.3a. For larger program versions Tarantula requires a developer to inspect more ranked statements on average to find a fault location. Nevertheless, there are four larger versions where Tarantula still performs comparatively good. The Ochiai ranking as seen in Figure 5.3b does not differ very much for the smaller program versions ($<$ 50.000 LOC) compared to Tarantula. For



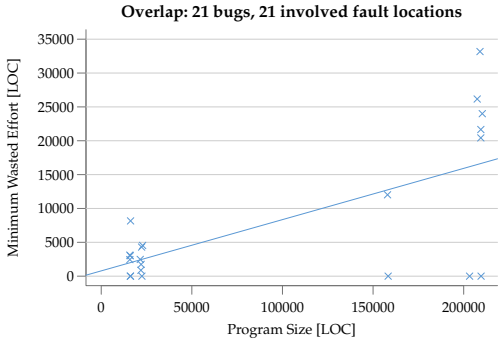
(a) Minimum wasted effort for the different examined program sizes using the Tarantula metric



(b) Minimum wasted effort for the different examined program sizes using the Ochiai metric



(c) Minimum wasted effort for the different examined program sizes using the Ample metric



(d) Minimum wasted effort for the different examined program sizes using the Overlap metric

Figure 5.3. Wasted effort of different program sizes for selected ranking metrics.

the larger program versions the Ochiai ranking metric performs different. Ochiai does not detect any prominent faults within the first 5.000 LOC, but detects most of the prominent faults around 10.000 to 15.000 LOC.

Although the average lines of code needed to find the prominent bug rise with the program size for the Tarantula and the Ochiai ranking metric, the program size does not have an influence on the wasted effort score for the Ample ranking metric as seen in Figure 5.3c. Ample consistently locates nearly all bugs with less than 10.000 inspected lines of code with the average being around 5.000 LOC. The results for Ample look promising compared to the other ranking metrics, but most prominent bugs can be found between 5.000 LOC and 10.000 LOC of examined lines which is practically unfeasible. Nevertheless, the Ample metric is the only metric of all 33 examined ranking metrics (see Section 2.3.3

for all metrics) where the program size has nearly no impact on the wasted effort metric. In contrast to that, the program size has the strongest impact of all examined ranking metrics on the Overlap metric as shown in Figure 5.3d. More than half of the larger program versions (> 150.000 LOC) require more than 20.000 ranked statements to be examined before finding the prominent fault location.

The data suggests, that for most ranking metrics there is a positive correlation between the number of ranked elements that need to be inspected and the program size. Interestingly, this correlation is very little for for the Ample ranking metric which leads to a very consistent performance for the Ample ranking metric across all examined buggy versions. To create SBFL techniques that are feasible for developers future research should focus on creating SBFL techniques that are independent from the program size and then push the average number of elements required to inspect down to an acceptable value.

Observation 2. Most ranking metrics show an increasing average wasted effort for larger program versions. Though there are ranking metrics where the program size does not seem to have a strong impact on the average wasted effort.

5.2 Research Question 2: How many bugs can be found when examining a fixed amount of ranked elements?

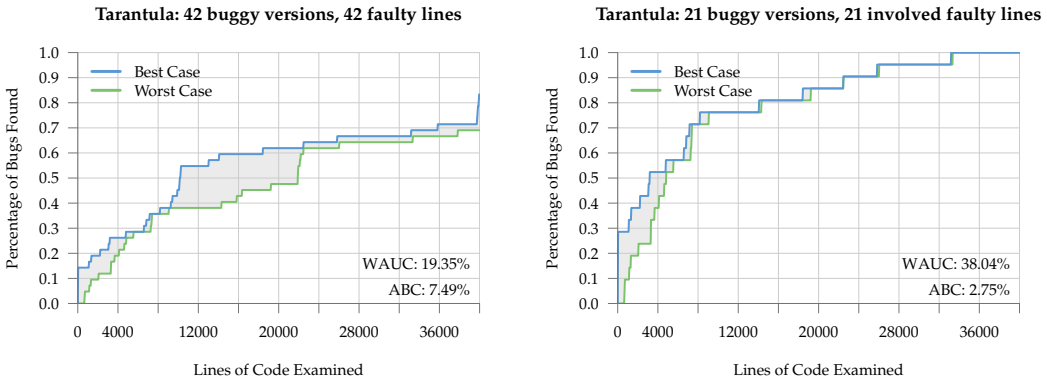
With real-world software systems having an increasingly larger size it is important to quickly guide developers to the actual fault location in case of an occurring bug. As reading code and manually classifying whether a given program element is indeed the fault location takes a lot of time for a developer, it is very important that automated debugging tools do not point a developer to too many suspicious program elements, as the time for classifying each of these sums up. In fact, Parnin and Orso [2011] have found in their study that developers fall back to traditional debugging methods if they find no value in the automated debugging tools soon enough. In the end, it is very important that by using automated debugging techniques it does not take longer for a developer to find the actual fault location than it would take him by using traditional debugging techniques. In order to achieve this, automated debugging techniques should point a developer straight to the fault location within the first ranked lines and not to false-positive fault locations.

5.2.1 Fault is not Among the First Few Hundred Ranked Elements

A very common approach in the current research practice is to measure the wasted effort (see equation 3.5) of a faulty statement as the percentage of program elements that a developer has to inspect before finding the actual fault location. Another commonly used metric is defined as the percentage of bugs found when examining a given percentage of code (see equation 3.9). A lot of research effort is put into minimizing the wasted effort metric or maximizing the percentage of found bugs metric. Different algorithms are evaluated on various benchmark suites under different circumstances and all seem to improve those metrics. However, spectrum-based fault localization has not been broadly applied by developers yet. Parnin and Orso [2011] have studied how developers use SBFL in practice and have shown that the current state of the art of SBFL is not very useful in practice. According to them, to make SBFL more useful it is of high importance to improve the absolute rank of faulty statements rather than percentage ranks.

Figure 5.4a shows the percentage of bugs found using the Tarantula ranking metric after examining a certain amount of lines of code. The horizontal axis represents the number of ranked elements inspected and the vertical axis shows the percentage of all fault locations that can be found. When trying to locate 50% of all fault locations nearly $20,000$ lines of code need to be inspected in the worst case. When only trying to locate 10% of all fault locations a developer is still required to inspect around $4,000$ lines of code using the Tarantula technique. These two numbers clearly indicate that the state of the art of SBFL is not feasible for a developer.

In figure 5.4b only fault locations that are involved at least once in a passing or failing test case are examined. Under this condition a developer needs to inspect less than half the elements compared to figure 5.4a to find the same proportion of fault locations. 50% of the



(a) Absolute fault localization performance of all available bugs

(b) Absolute fault localization performance of all bugs, that were involved in at least one execution trace

Figure 5.4. Absolute Tarantula fault localization performance

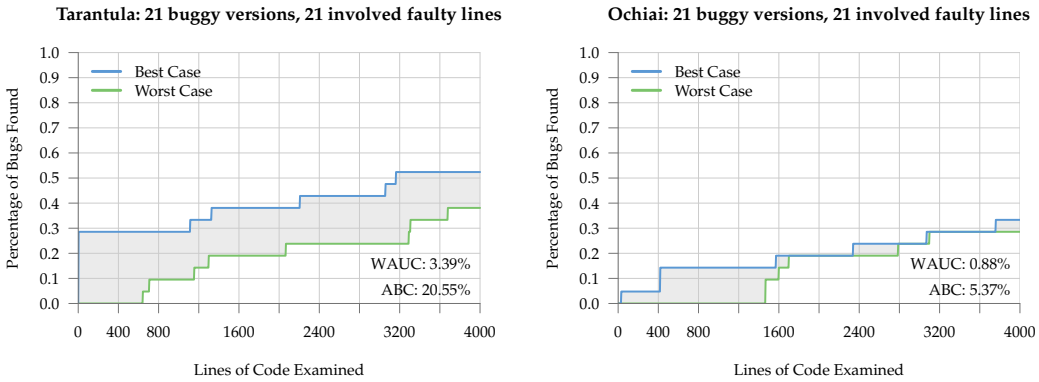
bug locations can be found after examining around 8,000 lines of code and 10% of the bug locations can be found after examining around 2,000 lines of code. The required number of lines of code to be inspected is still far too high for a developer to use the technique in a real-world scenario. Nevertheless, if all fault locations are involved in at least one execution SBFL is able to generate a significantly better ranking.

Observation 3. Fault locations that are involved in at least one failing or passing test case are ranked significantly higher. However, to locate more than 50% of all bugs even for involved faults a developer is required to inspect more than 4000 ranked elements on average.

5.2.2 Number of Files Investigated

The Number of Files Investigated metric defines the number of files a developer looks at, until the prominent bug is found. As defined in Section 3.4.3 it is assumed that the developer opens each file only once, inspects the complete file and then skips all program elements in the ranking where he already investigated the file the program element is contained in.

Figure 5.6 plots the average number of files a developer has to investigate prior to finding the file containing the prominent bug for all buggy versions where the prominent bug was involved in at least one execution trace. The vertical axis contains an entry for each ranking metric as defined in section 2.3.3. The horizontal axis represents the average number of files a developer has to investigate and is plotted on a logarithmic scale. The



(a) Tarantula fault localization performance

(b) Ochiai fault localization performance

Figure 5.5. Fault localization performance of the first 4000 lines of code

minor ticks mark the arithmetic mean between the next lower and higher major ticks. The logarithmic scale allows to spot the differences for the buggy versions where less than 100 files have to be investigated while still allowing to show the large part of the data where more than 100 files have to be investigated, and thus emphasizes the data that is most important to developers: the performance of the first investigated files. Each line in the plot represents the average number of files that need to be investigated for a specific buggy version of AspectJ using the ranking produced by the according ranking metric. A longer line indicates that multiple prominent bugs can be localized with the exact same average number of files. The thick long line represents the median of the number of investigated files for all buggy versions examined for the according ranking metric. The dotted line in the background is the median of the medians of the ranking metrics. The density of the buggy versions are represented by the colored area, thus a larger colored area means that a larger number of prominent bugs can be found within the effort range. The ranking metrics are sorted from top to bottom by their median in ascending order.

Regardless of the ranking metric a developer has to investigate more than 20 files on average to find the first prominent bug in any buggy version. In order to find 50% of the bugs a developer has to investigate more than 100 files on average which is an unfeasible task in practice. For all ranking metrics, the bulk contribution to the colored area is around the median of the respective metric. That means that it is very likely to find a lot of bugs when examining an average number of files that is close to the median of the respective ranking metric. However the median of the medians of all ranking metrics is larger than 150 files that need to be investigated. In addition to that, it is interesting to note that the Ochiai coefficient performs worse than the Tarantula ranking metric. In fact, the Tarantula metric is the third best performing metric.

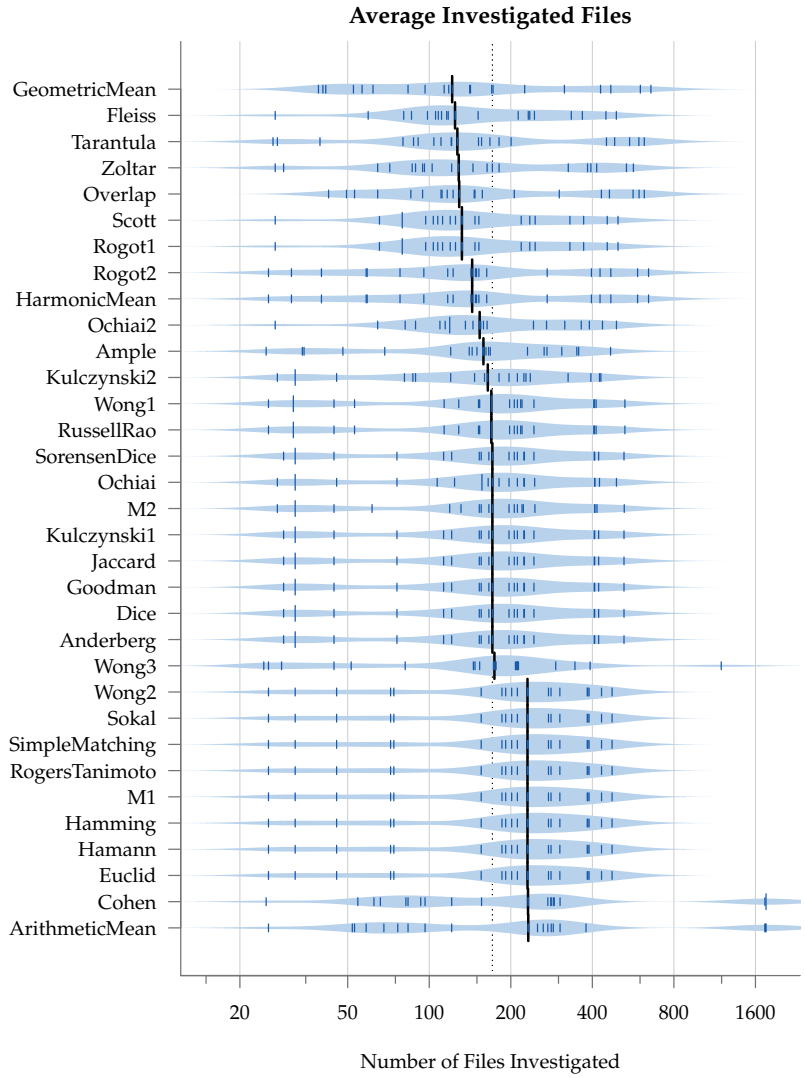


Figure 5.6. Average number of files investigated in order to find all prominent bugs that were involved at least once in a coverage trace for all different ranking metrics

The data suggests that the assumptions for the fault localization process of a developer stated in Section 3.4.3 do not offer any benefits. Even though the assumptions may come close to the real world behavior of developers (looking around the suspicious area and skipping lines in the ranking) the data suggests that the approach itself needs to be improved. Future work involves understanding the fault localization process employed by

real developers.

Observation 4. Developers need to inspect at least 20 different files to find the first bug using current SBFL algorithms. The median number of files to inspect is greater than 100 files for the best ranking metric. Most bugs can be found when examining a number of files that is close to the median of the corresponding ranking metric.

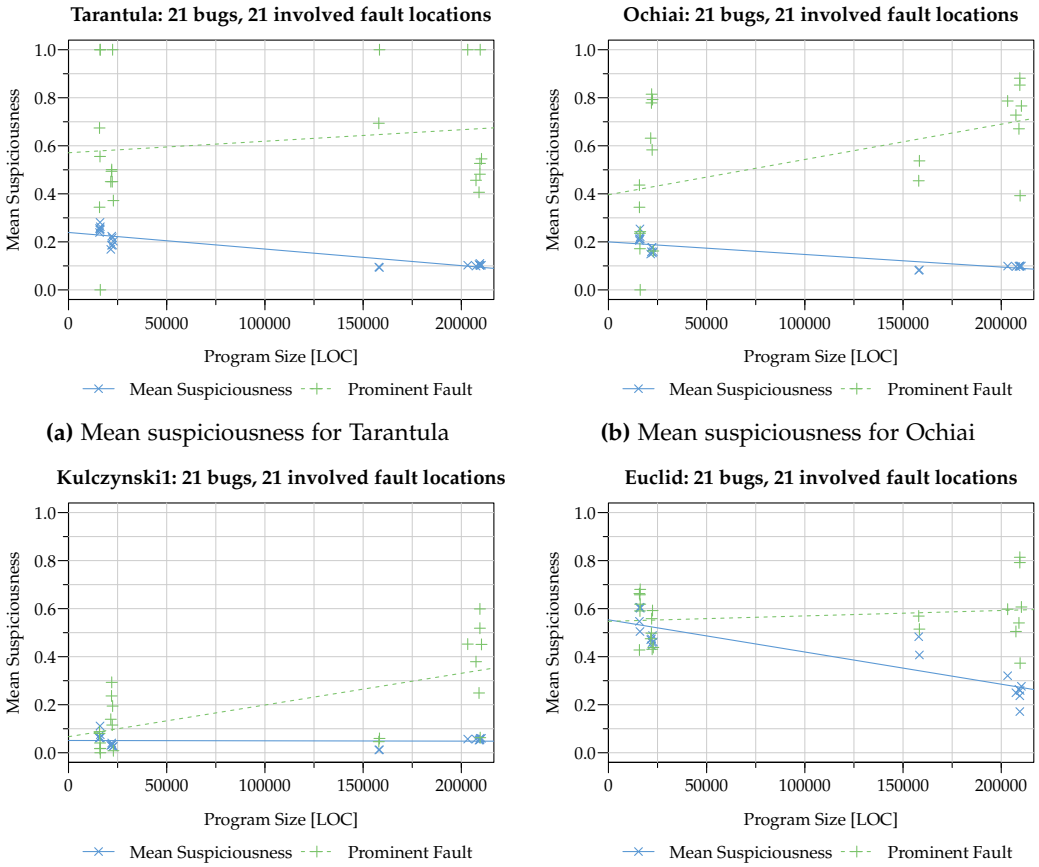
5.3 Research Question 3: How does the program size influence the suspiciousness scores produced by different ranking metrics?

The goal of SBFL is to guide a developer to statements that potentially contain a programming fault. As a suspiciousness score is assigned to each statement the score itself not only has an algorithmic meaning, but also may be presented to a developer for further interpretation. A suspiciousness score of a faulty element needs to differentiate the suspicious element from most of the unsuspicious program elements, such that the actual ranking position is not as important. DiGiuseppe and Jones [2014] have used the mean suspiciousness of all non-faulty elements compared to the suspiciousness of faulty elements as an indicator for how good faulty elements can stand out. If faulty program elements have a suspiciousness score near or below the mean suspiciousness score of non-faulty elements, there is no chance that a developer can distinguish the faulty elements from unsuspicious program elements.

Figure 5.7 shows the mean suspiciousness of all non-faulty elements and the suspiciousness of all prominent faults for the Tarantula, Ochiai, Kulczynski1 and Euclid ranking metric, respectively. The horizontal axis represents the program size in lines of code and the vertical axis represents the normalized suspiciousness as defined in Equation 3.18 on an interval from 0 (unsuspicious) to 1 (very suspicious). For each buggy version from the dataset the normalized mean suspiciousness of all non-faulty elements and the normalized suspiciousness of the prominent fault is plotted. The solid line is a linearly fitted line for the mean suspiciousness and the dashed line a linearly fitted line for the suspiciousness of the prominent fault.

Figure 5.7a displays the data for the Tarantula ranking metric. The mean suspiciousness for non-faulty elements is slightly decreasing with ~ 0.1 suspiciousness points for larger program versions. In addition to that, for larger program versions the deviation of the mean suspiciousness from the fit line is less than for smaller versions. Tarantula assigns five buggy versions with the highest suspiciousness score possible and most of the buggy versions with a suspiciousness score between 0.4 and 0.6. There is one buggy version with a suspiciousness score below the mean suspiciousness of all non-faulty elements. For larger program versions there is a slight upwards trend of ~ 0.1 suspiciousness points for the suspiciousness scores of the faulty elements. In general, the Tarantula ranking metric produces consistent results for smaller and larger program versions.

The Ochiai ranking metric as shown in Figure 5.7b shows a similar behavior compared to Tarantula for the mean suspiciousness. The mean suspiciousness slightly decreases and has smaller deviations for larger program versions. In contrast to Tarantula, the suspiciousness scores of the faulty elements of the Ochiai ranking metric do have a decent upwards trend of ~ 0.3 suspiciousness points for larger program versions. Both effects contribute to the high deviation between the mean suspiciousness and the suspiciousness scores of faulty elements for larger program versions. For smaller program versions the



(a) Mean suspiciousness for Tarantula

(b) Mean suspiciousness for Ochiai

(c) Mean suspiciousness for Kulczynski1

(d) Mean suspiciousness for Euclid

Figure 5.7. Mean suspiciousness for different ranking metrics

Ochiai ranking metric assigns five faulty versions a suspiciousness score below or near the mean suspiciousness. Nevertheless, the highest suspiciousness of 1.0 was assigned to no faulty element, in fact, all assigned suspiciousness scores of the faulty elements are smaller than 0.9. The results indicates that the program size influences the suspiciousness scores produced by the Ochiai ranking metric. For larger program versions there is a positive influence which means that the Ochiai ranking performs better for larger program versions than other ranking metrics.

The Kulczynski1 ranking metric as seen in Figure 5.7c is the only ranking metric of all examined metrics where the mean suspiciousness of all non-faulty elements has a horizontal fit line. This is interesting, as all other ranking metrics have a slight decreasing trend towards larger program versions. The program size thus has no influence on the

mean suspiciousness. Nevertheless the ranking metric assigns a suspiciousness score below or near to the mean suspiciousness to eight faulty versions. The suspiciousness scores of faulty elements rise similar to the scores the Ochiai ranking metric assigns. Note that the suspiciousness scores of the faulty elements all stay below 0.6. As the scores are normalized, this means that for every buggy version there are also non-faulty statements that score the highest suspiciousness of 1.0.

Figure 5.7d shows the Euclid ranking metric which shows the largest influence of the program size across all examined ranking metrics. The mean suspiciousness decreases by ~ 0.3 suspiciousness points for larger program versions. Compared to the other ranking metrics, the mean suspiciousness scores for all buggy versions show a larger deviation when compared to each other. Interestingly, the suspiciousness scores of the faulty statements do have a very small upwards trend similar to the Tarantula ranking metric, although the scores are not as scattered as by the Tarantula ranking metric. Mainly for smaller program versions there are eight faulty elements that scored a suspiciousness below or close to the mean suspiciousness.

All in all, the influence of the program size on the suspiciousness scores depends on the used ranking metric and may yield in various different behaviors. The data suggests that most ranking metrics have a lower mean suspiciousness score for larger program versions than for smaller program versions. In addition to that, the mean suspiciousness scores do not show a high variance across all examined ranking metrics. The suspiciousness scores of the faulty elements vary across the different ranking metrics, not only in the score itself, but also in the distance to the mean suspiciousness scores of program versions with the same size as the respective faulty elements.

Future research may leverage the diversity of the different ranking metrics to combine the scores for better results. In order to achieve that it is necessary to better understand the influence of the problem domain size on the ranking metrics.

Observation 5. The program size does influence the suspiciousness scores, but the influence mainly depends on the ranking metric. A pattern observed across all ranking metrics is that the mean suspiciousness score does not rise for larger program versions.

5.4 Research Question 4: Are the fault localization performance metrics currently used by the research community valid?

The research community currently uses the Wasted Effort metric (see Section 3.3.1) and the Proportion of Localized Bugs metric (see Section 3.3.2) as benchmark metrics to measure the performance of SBFL techniques [DiGiuseppe and Jones, 2014; Naish et al., 2011; Abreu et al., 2006; Lucia et al., 2014]. In a human case study with developers Parnin and Orso [2011] have shown that the behavior of developers is not reflected by the model assumptions made for the performance metrics. This could mean that the currently used performance metric values do *not* correlate with the debugging effort in practice and hence, the used performance metrics may not be valid metrics for the debugging effort in practice. This research questions aims to find evidence in a large real-world project to either support or contradict the validity of the currently used performance metrics.

5.4.1 Hit@X

Considering the Hit@X metric (Equation 3.11)), a developer might find it feasible to manually scan a list of 100 ranked program elements. Figure 5.8 shows the Hit@100 metric for all examined SBFL ranking metrics. The vertical axis represents the number of bugs found after examining the first 100 ranked elements produced by each ranking metric, respectively. The plot marks the best-, average and worst-case for each ranking metric. For the Hit@100 metric the average and worst-case are all equal to zero, which means that no ranking metric can find any bug within the first 100 ranked lines of code in the average or worst-case. In the best case, half of the ranking metrics are able to locate at least one bug, Overlap and Tarantula may find up to six bugs. The data suggests that investigating only the first 100 ranked elements does not help the developer at all. Nevertheless, interpreting and investigating 100 ranked elements is already very much work for a developer.

Figure 5.9 shows the Hit@1000 metric for all ranking metrics, that means 10 times more ranked elements are included in the ranking. In the best-case all ranking metrics find at least one of the 21 prominent fault locations. With seven bugs found, Overlap has the highest number of bugs found in the best-case while the average and worst-case of Overlap remain zero. This can be explained by the low decidedness of Overlap introducing large randomly ordered parts in the ranking as seen in Section 5.1. After Overlap, GeometricMean and Tarantula are able to locate six prominent bugs in the best case. In addition to that, GeometricMean is able to find the most bugs in the average case compared to the other ranking metrics. In fact, for GeometricMean the average case is equal to the best-case when examining the first 1000 ranked elements. The average case of the HarmonicMean and Rogot2 ranking metric locate one fault less than GeometricMean in the average case. The three metrics have in common that the average and best-case are identical and in the worst case the three metrics can locate the most bugs compared to all

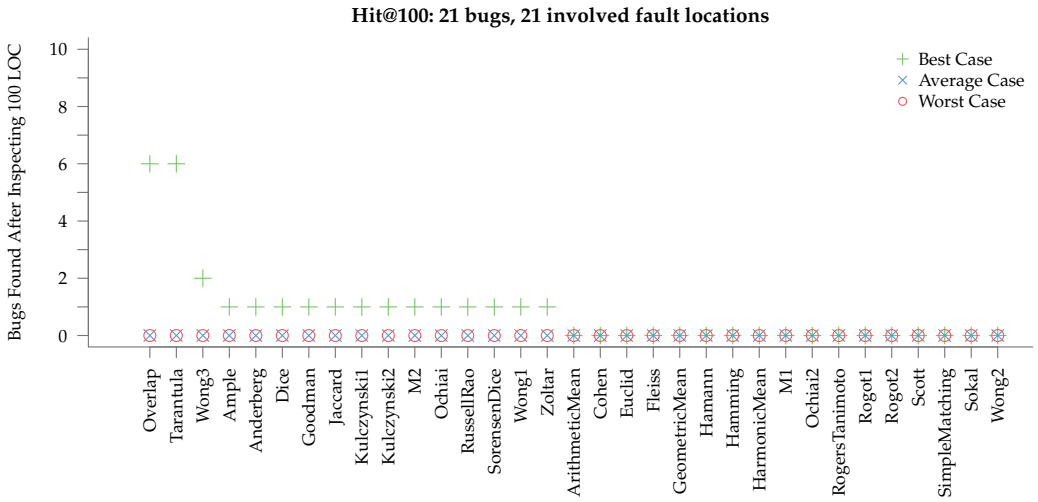


Figure 5.8. Number of bugs found after examining the first 100 ranked lines. The ranking metrics are ordered descending by the best case and alphabetically in case of a tie.

other metrics except Ample. Note that the four formulas with the highest number of bugs found in the worst-case (Ample, GeometricMean, HarmonicMean, Rogot2) all have other very similar ranking metrics when comparing the arithmetic operations, but which perform comparatively poorly (Wong2, Ochiai2, ArithmeticMean, Kulzcynski2 respectively).

For example, the difference between GeometricMean and Ochiai2 is only the additional term $-n_{nf} \cdot n_{ip}$ which is added to the numerator in the GeometricMean ranking metric. However, Ochiai2 can only find a single bug within the first 1000 ranked elements, whereas GeometricMean finds at least four bugs. The semantic meaning of the additional term is that it lowers the suspiciousness for program elements which are involved in passing test cases or not involved in failing test cases. As this affects a large proportion of the program elements, it could be the explanation for the performance difference.

All in all, when examining the first 1000 ranked elements $\sim$ 60% of the ranking metrics guarantee to find at least one bug. The median of the number of found bugs across all ranking metrics is three. Nevertheless, 1000 ranked elements are too many elements for a developer to examine in practice while in the best-case it is only possible to find a third of all prominent bugs. In addition to that, the plot contains only prominent bugs that are involved in at least one test case. When adding all non-involved prominent bugs to the dataset the plot is exactly the same, which implies that no bug that is not involved can be found within the first 1000 ranked elements.

Future work includes understanding the differences between the ranking metrics and their impact on fault localization performance. Furthermore, future research should develop SBFL techniques that are capable of reliably finding bugs in the first 100 ranked elements,

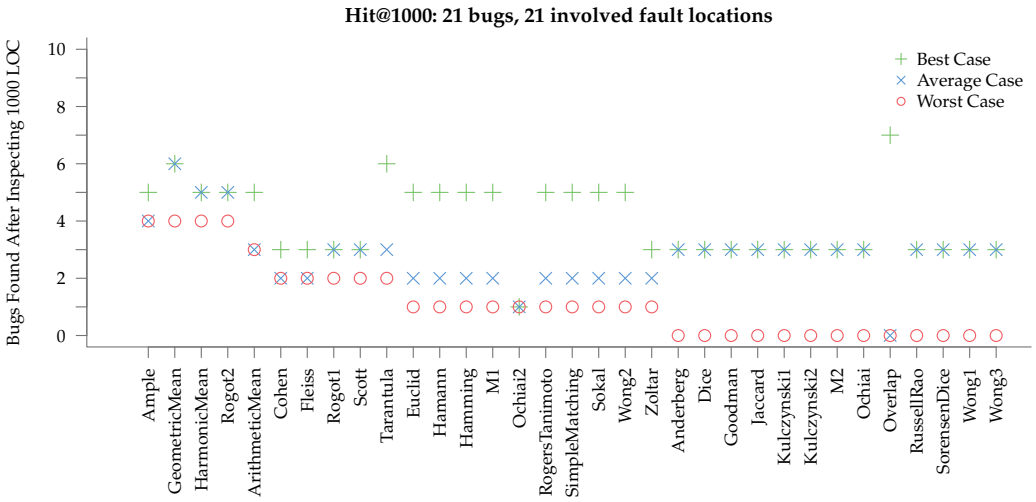


Figure 5.9. Number of bugs found after examining the first 1000 ranked lines. The ranking metrics are ordered descending by the worst case and alphabetically in case of a tie.

as those are the elements where it is feasible for developers to look for suspicious elements.

Observation 6. Applying the Hit@100 metric not a single bug was found in the average case by any ranking metric. Up to a third of all bugs could be found in the best-case when examining the Hit@1000 metric. The median number of bugs found in the average case across all ranking metrics when examining 1000 ranked elements is less than 15%.

5.4.2 Performance Metrics in Comparison

Table 5.1 shows various performance metrics for all examined ranking metrics. All values are percentages rounded to two digits and were computed as defined in Chapter 3. The best value of each performance metric is printed in bold font.

ABC (defined in Equation 3.20) refers to the Area Between Two Curves, namely the area between the *min_pbl* and *max_pbl* curves. The $WAUC(\{min, avg, max\}\_pbl)$ metrics use the weighting function W_8 as defined in Equation 3.22 to compute the weighted area under the respective proportion of bugs localized curve. For the $\{min, avg, max\}\_we\_aggregated$ metrics, the wasted effort metric of all bugs has been aggregated using the arithmetic mean. The last six columns show the values for the proportion of localized bugs (see Equation 3.9) metric for 1% of examined code and 10% of examined code. The last row of the table contains the arithmetic mean of each performance metric for all ranking metrics.

Naish et al. [2011] report a mean value of 12.27% for the $avg\_pbl_{0.01}$ metric using programs from the SIR and 23 different ranking metrics. In addition to that, a mean value

of 46.57% is achieved, when investigating 10% of the code using the same artifacts. While for 1% of examined code, the mean PBL value in this study is lower (3.32%), the mean value for 10% of examined code is slightly higher (57.14%). Lucia et al. [2014] report a mean value of 36.01% for the $avg\_pbl_{0.1}$ metric using 23 different ranking metrics and projects from the SIR. The mean values indicate that the performance using the PBL metric does not differ in an unexpected way when comparing the results of this study with the results of other researchers with smaller programs from the SIR.

For the wasted effort metric, Abreu et al. [2009b] report a mean value of 9.86% using four ranking metrics. Abreu et al. [2006] measured a mean value of 12.5% for four ranking metrics, while Naish et al. [2011] report a mean value for the metric of 22.25% with 25 different ranking metrics. All reported mean values originate from a dataset containing only programs from the SIR. The mean value of the average wasted effort metric in this study is 20.52%. This also indicates that the WE metric for this study compared to the results of other researchers using smaller programs does not differ in an unexpected way.

The previous section examining the Hit@100 and Hit@1000 metric has shown that the performance of SBFL for a project of this size is too unreliable for a developer to adapt SBFL in practice, as no developer will examine a list of ranked program elements longer than 1000 elements even when embracing skipping and zig-zag-patterns to scan the list. Hence, even though the values of commonly used performance metrics computed in this study fit into the bounds of common research results, the rankings produced by SBFL in this study are unusable for developers in practice. This means that the used performance metrics are not suited to distinguish between useful and impractical rankings, thus the data provides evidence that the commonly used performance metrics are not valid.

Best Ranking Metrics

A ranking metric has the best result for a performance metric if the value in Table 5.1 is written in bold font. There are noteworthy trends when comparing the commonly used performance metrics with the performance metrics proposed in this study.

For example, the ranking metrics that score best for all three $\{min|avg|max\}\_pbl_{0.1}$ values are: Euclid, Hamann, Hamming, M1, RogersTanimoto, SimpleMatching, Sokal and Wong2. The ABC values of the mentioned ranking metrics are worse than the average ABC value of all ranking metrics. In addition to that, most $WAUC$ values of the mentioned ranking metrics are below the average $WAUC$ value of all ranking metrics. However, the reverse direction does not hold as for example Overlap has a very bad ABC value but also a PBL value below the mean PBL value.

Ranking metrics with a low ABC value ($< 2.0\%$), namely Ample, ArithmeticMean, Cohen, GeometricMean, HarmonicMean, and Rogot2 tend to find more bugs in the first percentage of examined code than the other ranking metrics. Next to that, ranking metrics, with an average WAUC value greater than 48% (Ample, Fleiss, GeometricMean, HarmonicMean, Rogot2, Tarantula) have less than or nearly average PBL scores.

While these trends are only the result of an anecdotal investigation, there do seem to be

Table 5.1. Various SBFL performance metrics for the different ranking metrics. The performance metrics were calculated using 21 involved prominent bugs.

Ranking metric	*ABC*	*WAUC(min_pbl)*	*WAUC(avg_pbl)*	*WAUC(max_pbl)*	*min_we_aggregated*	*avg_we_aggregated*	*max_we_aggregated*	$min\_pbl_{0.01}$	$avg\_pbl_{0.01}$	$max\_pbl_{0.01}$	$min\_pbl_{0.1}$	$avg\_pbl_{0.1}$	$max\_pbl_{0.1}$
Ample	**1.62**	46.99	48.38	49.76	15.82	16.63	17.45	9.52	9.52	14.29	57.14	57.14	57.14
Anderberg	3.90	39.96	42.79	45.62	15.97	17.92	19.87	0.00	0.00	4.76	57.14	57.14	57.14
ArithmeticMean	1.70	38.98	40.66	42.33	38.96	39.81	40.66	9.52	9.52	9.52	47.62	47.62	47.62
Cohen	1.69	39.27	40.83	42.40	39.15	40.00	40.84	**14.29**	**14.29**	14.29	42.86	47.62	47.62
Dice	3.90	39.96	42.79	45.62	15.97	17.92	19.87	0.00	0.00	4.76	57.14	57.14	57.14
Euclid	7.92	38.67	44.82	50.97	22.45	26.41	30.37	0.00	0.00	4.76	**61.90**	**66.67**	**71.43**
Fleiss	4.52	46.63	48.35	50.07	17.11	19.38	21.65	4.76	4.76	4.76	47.62	47.62	47.62
GeometricMean	1.70	**48.19**	**50.55**	52.92	11.91	**12.76**	**13.62**	9.52	9.52	9.52	52.38	52.38	57.14
Goodman	3.90	39.96	42.79	45.62	15.97	17.92	19.87	0.00	0.00	4.76	57.14	57.14	57.14
Hamann	7.92	38.67	44.82	50.97	22.45	26.41	30.37	0.00	0.00	4.76	**61.90**	**66.67**	**71.43**
Hamming	7.92	38.67	44.82	50.97	22.45	26.41	30.37	0.00	0.00	4.76	**61.90**	**66.67**	**71.43**
HarmonicMean	**1.62**	46.45	48.66	50.87	12.80	13.61	14.42	9.52	**14.29**	19.05	52.38	52.38	52.38
Jaccard	3.90	39.96	42.79	45.62	15.97	17.92	19.87	0.00	0.00	4.76	57.14	57.14	57.14
Kulczynski1	3.90	39.96	42.79	45.62	15.97	17.92	19.87	0.00	0.00	4.76	57.14	57.14	57.14
Kulczynski2	2.05	42.78	46.09	49.39	13.16	14.19	15.22	0.00	0.00	4.76	**61.90**	61.90	61.90
M1	7.92	38.67	44.82	50.97	22.45	26.41	30.37	0.00	0.00	4.76	**61.90**	**66.67**	**71.43**
M2	3.89	40.05	42.92	45.79	16.14	18.10	20.05	0.00	0.00	4.76	57.14	61.90	61.90
Ochiai	2.14	40.23	43.04	45.85	15.57	16.65	17.73	0.00	0.00	4.76	57.14	57.14	57.14
Ochiai2	2.08	44.20	45.15	46.11	14.64	15.68	16.73	4.76	4.76	4.76	52.38	52.38	52.38
Overlap	9.44	36.23	45.01	53.79	**11.17**	15.89	20.61	0.00	0.00	**28.57**	28.57	42.86	47.62
RogersTanimoto	7.92	38.67	44.82	50.97	22.45	26.41	30.37	0.00	0.00	4.76	**61.90**	**66.67**	**71.43**
Rogot1	3.89	46.14	47.05	47.95	16.27	18.22	20.16	9.52	9.52	9.52	47.62	47.62	47.62
Rogot2	**1.62**	46.45	48.66	50.87	12.80	13.61	14.42	9.52	**14.29**	19.05	52.38	52.38	52.38
RussellRao	5.09	39.81	43.20	46.58	15.60	18.15	20.69	0.00	0.00	4.76	57.14	61.90	61.90
Scott	3.89	46.14	47.05	47.95	16.27	18.22	20.16	9.52	9.52	9.52	47.62	47.62	47.62
SimpleMatching	7.92	38.67	44.82	50.97	22.45	26.41	30.37	0.00	0.00	4.76	**61.90**	**66.67**	**71.43**
Sokal	7.92	38.67	44.82	50.97	22.45	26.41	30.37	0.00	0.00	4.76	**61.90**	**66.67**	**71.43**
SorensenDice	3.90	39.96	42.79	45.62	15.97	17.92	19.87	0.00	0.00	4.76	57.14	57.14	57.14
Tarantula	2.54	44.33	49.15	**53.97**	12.64	13.91	15.17	0.00	0.00	**28.57**	47.62	47.62	52.38
Wong1	5.09	39.81	43.20	46.58	15.60	18.15	20.69	0.00	0.00	4.76	57.14	61.90	61.90
Wong2	7.92	38.67	44.82	50.97	22.45	26.41	30.37	0.00	0.00	4.76	**61.90**	**66.67**	**71.43**
Wong3	9.04	38.80	43.71	48.62	16.08	20.60	25.12	0.00	0.00	14.29	**61.90**	61.90	61.90
Zoltar	2.07	41.91	44.96	48.02	13.77	14.81	15.84	9.52	9.52	14.29	47.62	47.62	57.14
Mean value	4.62	48.52	44.91	41.29	18.21	20.52	22.83	3.03	3.32	8.80	54.98	57.14	59.02

unknown variables the research community has not yet investigated. In fact, the Ochiai ranking metric which is commonly referred to as a good ranking metric has an average performance on this large project. In addition to that, the Tarantula metric has a very solid performance throughout this study. All in all, the data supports that the validity of the commonly used performance metrics has to be questioned. Additional performance metrics should be developed, improved and thoroughly validated, for example as described

by Schneidewind [1992].

Observation 7. Even though a maximum of seven of 21 bugs can be found in the best case when investigating the 1000 most suspicious program elements, the commonly used performance metrics are within the bounds of the results of other researchers using the SIR. Thus, the commonly used performance metrics do not indicate that the results of this study are unfeasible in practice.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

This study examines the impact of a large program size on SBFL performance. Several ranking and performance metrics have been considered and compared to determine the fault localization performance. In addition to that, the study evaluates the validity of commonly established performance metrics and proposes enhanced performance metrics for SBFL.

The investigated experiments provide evidence that it is difficult to apply SBFL to large projects. First, a large code base with a comparatively small number of executed test cases leads to randomness in the produced ranking as multiple elements share the same suspiciousness. The *ABC* metric has been proposed to measure the degree of randomness induced in a ranking. In addition to that, the program size can have a strong impact on the wasted effort metric. In general, a larger program size increases the wasted effort, although there exist ranking metrics where this influence is rather small. Besides that, the data reveals that the suspiciousness scores assigned to all program elements can also be influenced by the program size. While the influence mainly depends on the ranking metric, the mean suspiciousness across all non-faulty elements does not rise with larger program versions, but can decline.

When examining a fixed amount of ranked program elements, fault locations that are involved in at least one execution are ranked considerably higher. Still, even for good ranking metrics it is required to inspect at least 4000 lines of code or at least 100 files to find half of the fault locations. This makes SBFL unfeasible for large projects in practice, even when embracing skipping and zig-zagging patterns to scan the list of ranked elements.

The data also suggests that the validity of commonly established performance metrics has to be questioned. The values of the common performance metrics are within the bounds of values that other researchers have measured with projects from the SIR. Yet, the data of this study suggests that the SBFL performance is unfeasible in practice, as only a third of all involved prominent bugs can be found when examining the most suspicious 1000 ranked elements. Hence, the commonly used performance metrics do not indicate the practical unfeasibility of SBFL for large projects and are thus not valid performance metrics.

6.2 Future Work

Future work includes the evaluation of the concerns raised in this study with enriched data sets to eliminate the single project bias and to validate them using a thorough empirical study. In addition to that the behavior of developers should further be examined to build a solid model from their debugging behavior. The proposed performance metrics can then be validated and refined to better reflect the debugging model. It may also be helpful to consider additional data to compute the suspiciousness scores to eliminate that multiple elements share the same suspiciousness and to diversify the suspiciousness scores to better reflect the actual suspiciousness of a program element.

Bibliography

[Asp] AspectJ language extension. URL http://www.eclipse.org/aspectj/.

[Abreu et al. 2006] R. Abreu, P. Zoeteweij, and A. J. Van Gemund. An evaluation of similarity coefficients for software fault localization. In *Dependable Computing, 2006. PRDC'06. 12th Pacific Rim International Symposium on*, pages 39–46. IEEE, 2006.

[Abreu et al. 2009a] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. C. van Gemund. A practical evaluation of spectrum-based fault localization. *J. Syst. Softw.*, 82(11):1780–1792, Nov. 2009a. ISSN 0164-1212. doi: 10.1016/j.jss.2009.06.035. URL http://dx.doi.org/10.1016/j.jss.2009.06.035.

[Abreu et al. 2009b] R. Abreu, P. Zoeteweij, and A. J. Van Gemund. Spectrum-based multiple fault localization. In *Automated Software Engineering, 2009. ASE'09. 24th IEEE/ACM International Conference on*, pages 88–99. IEEE, 2009b.

[Britton et al. 2013] T. Britton, L. Jeng, G. Carver, P. Cheak, and T. Katzenellenbogen. Reversible debugging software. Technical report, Technical report, University of Cambridge, Judge Business School, 2013.

[Casanova et al. 2011] P. Casanova, B. Schmerl, D. Garlan, and R. Abreu. Architecture-based run-time fault diagnosis. In *Proceedings of the 5th European Conference on Software Architecture*, ECSA'11, pages 261–277, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-23797-3. URL http://dl.acm.org/citation.cfm?id=2041790.2041827.

[Dallmeier and Zimmermann 2007] V. Dallmeier and T. Zimmermann. Extraction of bug localization benchmarks from history. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, pages 433–436, New York, NY, USA, 2007. ACM, ACM. ISBN 978-1-59593-882-4. doi: 10.1145/1321631.1321702. URL http://doi.acm.org/10.1145/1321631.1321702.

[DiGiuseppe and Jones 2014] N. DiGiuseppe and J. Jones. Fault density, fault types, and spectra-based fault localization. *Empirical Software Engineering*, pages 1–40, 2014. ISSN 1382-3256. doi: 10.1007/s10664-014-9304-1. URL http://dx.doi.org/10.1007/s10664-014-9304-1.

[Do et al. 2005] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405–435, 2005.

[Harrold et al. 2000] M. J. Harrold, G. Rothermel, K. Sayre, R. Wu, and L. Yi. An empirical investigation of the relationship between spectra differences and regression faults. *Software Testing Verification and Reliability*, 10(3):171–194, 2000.

[Herzig et al. 2013] K. Herzig, S. Just, and A. Zeller. It's not a bug, it's a feature: How misclassification impacts bug prediction. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 392–401. IEEE Press, 2013.

[Jones 2008] J. A. Jones. *Semi-automatic fault localization*. PhD thesis, Georgia Institute of Technology, 2008.

[Jones et al. 2002] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th international conference on Software engineering*, pages 467–477. ACM, 2002.

[Korel and Laski 1988] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, 1988.

[Lucia et al. 2014] Lucia, D. Lo, and X. Xia. Fusion fault localizers. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 127–138, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3013-8. doi: 10.1145/2642937.2642983. URL http://doi.acm.org/10.1145/2642937.2642983.

[Naish et al. 2011] L. Naish, H. J. Lee, and K. Ramamohanarao. A model for spectra-based software diagnosis. *ACM Transactions on software engineering and methodology (TOSEM)*, 20(3):11, 2011.

[Openhub 2014a] Openhub. The aspectj open source project on open hub : Commits summary page, 2014a. URL https://www.openhub.net/p/freshmeat_aspectj/commits/summary. [Online; accessed 29-September-2014].

[Openhub 2014b] Openhub. The aspectj open source project on open hub : Languages page, 2014b. URL https://www.openhub.net/p/freshmeat_aspectj/analyses/latest/languages_summary. [Online; accessed 29-September-2014].

[Parnin and Orso 2011] C. Parnin and A. Orso. Are automated debugging techniques actually helping programmers? In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 199–209, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0562-4. doi: 10.1145/2001420.2001445. URL http://doi.acm.org/10.1145/2001420.2001445.

[Schneidewind 1992] N. F. Schneidewind. Methodology for validating software metrics. *Software Engineering, IEEE Transactions on*, 18(5):410–422, 1992.

[Weiser 1981] M. Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.

[Wong and Debroy 2010] W. E. Wong and V. Debroy. Software fault localization. *Encyclopedia of Software Engineering*, 1:1147–1156, 2010.

Declaration

I hereby declare that the work presented in this thesis is entirely my own. I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature