Code Coverage Measurement and Fault Localization Approaches

A Thesis Submitted for the Degree of
Doctor of Philosophy
of the University of Szeged

by

Ferenc Horváth

Supervisor:

Árpád Beszédes, Ph.D.
associate professor

Veritas
Virtus
Libertas

Doctoral School of Computer Science
Department of Software Engineering
Faculty of Science and Informatics
University of Szeged

Szeged, 2023

Preface

As I remember, my connection to science goes way back to my early childhood. Although I didn't have access to computers back then, I clearly remember that as a child I was constantly bombarding my parents with questions about everything. Also, to this day, every family photo session brings up a picture of me fiddling with my grandparents' radio when I was just a few years old. As time went on, my interest became more and more focused on how technical things around me worked. Another defining memory is that I played with LEGO endlessly. First, I just followed the assembly instructions, but later, I let my imagination run wild to create increasingly complex structures.

In primary school, my favourite subjects were always real subjects such as mathematics, physics, and chemistry. Then, as a teenager, when I was introduced to computers and computer science, I was immediately drawn to this world. Interestingly, I didn't get into programming until relatively late in life. It was in high school that I first learned how to write programs. But it was a very defining moment because from then on I felt more and more that it was what I really loved to do.

From there, it was a straight road to the university and later to becoming a Ph.D. student, where – as a software engineer – I can fulfill my desire to be creative and build complex systems, and – as a researcher – I can use my curiosity I had as a child, as well as the many years of experience I had acquired by constantly asking questions and wondering about how things work.

Ferenc Horváth, 2023

Contents

List of Tables

List of Figures

1 Introduction

Code coverage measurement plays an important role in white-box testing, both in industrial practice and academic research. Several areas are highly dependent on code coverage as well, including test case generation, test prioritization, fault localization, and others. Out of these areas, this dissertation focuses on two main topics, and the thesis points are divided into two parts accordingly. The first part consists of one thesis point that discusses the differences between methods for measuring code coverage in Java and the effects of these differences. The second part focuses on a fault localization technique called spectrum-based fault localization that utilizes code coverage to estimate the risk of each program element being faulty. More specifically, the corresponding two thesis points are discussing the improvement of the efficiency of spectrum-based approaches by incorporating external information, *e.g.,* users' knowledge, and context data extracted from call chains.

Put simply, code coverage is a test completeness measure that is used to express to what portion of the implemented functionality has been exercised in terms of the number of executed code elements during dynamic testing. One may argue, that if it is simply used as an overall completeness measure, minor inaccuracies of coverage data reported by a tool do not matter that much; however, in certain situations, they can lead to serious confusion. For example, a code element that is falsely reported as covered can introduce false confidence in the test, or it can misguide test case generation approaches.

During my work, I started looking into code coverage measurement issues for the Java programming language, when me and my colleagues noticed that certain mutation-based methods were behaving rather strangely. For Java, the prevalent approach to code coverage measurement is to use *bytecode instrumentation* due to its various benefits over *source code instrumentation* (instrumentation means placing probes into the program which will collect coverage information during runtime). However, as we experienced, bytecode instrumentation-based code coverage tools produce different results in terms of the reported items that are covered concerning source code instrumentation-based tools. Since most of the applications of code coverage operate on the source code, the latter category is treated as more precise, and deviations from it can lead to issues in the interpretation of the data.

This dissertation reports on an empirical study to compare the code coverage results provided by two tools for Java coverage measurement on method level (one for each instrumentation type). In particular, we want to find out how much a bytecode instrumentation approach is inaccurate compared to a source code instrumentation method. The differences are systematically investigated both in quantitative (how much the outputs differ) and in qualitative terms (what are the causes for the differences). In addition, the impact on test prioritization and test suite reduction, a possible application of coverage measurement, is investigated in more detail as well. We look at how smaller or greater differences in the coverage data itself influence the application: whether a small deviation in the coverage information causes a significant difference in the derived data or the opposite?

Fault localization is considered a difficult and time-consuming activity. Tool support for automated fault localization in program debugging is limited because state-of-the-art algorithms often fail to provide efficient help to the user. They usually offer a ranked list of suspicious code elements, but the fault is not guaranteed to be found among the highest ranks. In Spectrum-Based Fault Localization (SBFL) – which uses code coverage information of test cases and their execution outcomes to calculate the ranks –, the developer has to investigate several locations before finding the faulty code element. Yet, all the knowledge they a priori have or acquire during this process is not reused by the SBFL tool.

This dissertation proposes an approach in which the developer interacts with the SBFL algorithm by giving feedback on the elements of the prioritized list, called *Interactive Fault Localization (iFL)*. We exploit the contextual knowledge of the user about the next item in the ranked list (*e.g.,* a statement), with which larger code entities (*e.g.,* a whole function) can be repositioned in their suspiciousness. First, we evaluated the approach using simulated users incorporating two types of imperfections, their knowledge and confidence levels. Then, we empirically evaluated the effectiveness of the approach with real users in two sets of experiments: a quantitative evaluation of the successfulness of using *iFL*, and a qualitative evaluation of practical uses of the approach with experienced programmers.

In SBFL, program elements such as statements or functions are ranked according to a suspiciousness score which can guide the programmer in finding the fault more efficiently. However, such a ranking does not include any additional information about the suspicious code elements. Although there have been attempts to include control or data flow information in the process, these attempts did not succeed because of scalability issues to real programs and real faults. This dissertation proposes to complement function-level spectrum-based fault localization with *function call chains – i.e.,* snapshots of the call stack occurring during execution – on which the fault localization is first performed, and then narrowed down to functions. Experiments using medium-sized real programs show that the effectiveness of the process, in terms of localization expense, can be substantially improved concerning the basic function-level approach with a manageable computation overhead.

1.1 Challenges

Challenge 1: Accuracy of code coverage measurement (C1). Many software testing fields, like white-box testing, test case generation, test prioritization, and fault localization, depend on code coverage measurement. If used as an overall completeness measure, the minor inaccuracies of coverage data reported by a tool do not matter that much, however, in certain situations they can lead to serious confusion. For example, a code element that is falsely reported as covered can introduce false confidence in the test.

Challenge 2: Effects of code coverage differences (C2). When someone applies a certain code coverage measurement method in an industrial or experimental setting it is important to know how the chosen method influences the application. It is also crucial to know whether these discrepancies imply any risks in the concrete situation, and how these risks can be mitigated.

Challenge 3: Efficiency of fault localization (C3). Localizing faults in a program is a typically complex and hard task of software development. Many approaches aim to support the developers by automating different parts of the debugging process, however, state-of-the-art methods often fail to provide efficient help to the users. For example, it is not unusual that developers have to investigate several of the suggested locations in the code before finding the faulty element.

Challenge 4: User centric fault localization (C4). There are relatively few fault localization approaches that offer a seamless user experience. Most of the methods are limited to experimental scenarios, and existing tools are often complicated to use. In addition, there are hardly any tools that utilize the extra information which can be extracted from the interaction between the user and the tool.

1.2 Structure of the Dissertation

The contributions presented in this dissertation are organized into two parts based on their corresponding topic (code coverage measurement and fault localization) and are presented in three chapters that are mostly aligned with challenges from Section 1.1.

- **Thesis I** considers the several aspects of code coverage measurement for Java programs. It elaborates on what type of inaccuracies one might experience while using different measurement tools, and it presents a quantitative and qualitative analysis of these cases. In addition, it discusses how and to what extent the identified flaws of accuracy affect different applications like test case prioritization and test suite reduction.

  This thesis is presented in Chapter 3, that is organized as follows. Section 3.2 gives the background of code coverage and its usability in different applications, and lists the risks of coverage measurement for Java and the relation to similar works as well. Research aims are stated in more detail in Section 3.3. Section 3.4 describes the basic setup for the experiments, the tools, and the subject systems, while Section 3.5 presents the results of the empirical study. This section is organized according to our research agenda: first, we concentrate on the quantitative and qualitative differences, and then, we investigate the effect on the test case prioritization application. Finally, Section 3.6 summarizes our findings and provides a more general discussion before concluding in Section 3.7.

  Thesis I responds to Challenges 1 and 2 discussed in Section 1.1.

- **Thesis II** is about how the interactivity between the developers and a fault localization tool can be utilized to improve the effectiveness and efficiency of the process. It proposes a new approach in which the developers can interact with the fault localization algorithm by giving feedback on the suggested code elements. It presents the design and results of the experiments that empirically evaluate this approach with simulated and real users.

This thesis is presented in Chapter 4, that is organized as follows. Section 4.2 contains an example to illustrate our approach, which is then discussed in detail in Section 4.3. Section 4.4 summarizes the goals of the empirical assessment, while Sections 4.5 and 4.6 present the experimental results for the simulated and the real users, respectively. A discussion of our findings is presented in Section 4.7 with possible threats to validity in Section 4.8. Related works are overviewed in Section 4.9, before concluding with Section 4.10.

Thesis II responds to Challenges 3 and 4 discussed in Section 1.1.

- **Thesis III** studies how existing SBFL algorithms can be extended with additional information. It proposes a new approach which complements fault localization by utilizing snapshots of call stacks which occur during the execution of a program as extra information. Also, it describes how the corresponding experiments for empirical evaluation were constructed, and it shows the extent of the achieved improvements over traditional algorithms.

  This thesis is presented in Chapter 5, that is organized as follows. Related work is overviewed in Section 5.2, before presenting our approach in Section 5.3. Section 5.4 includes the description of the empirical study. The associated results are introduced in Section 5.5, before concluding in Section 5.6.

  Thesis III responds to Challenge 3 discussed in Section 1.1.

2
Background

2.1 Code Coverage Measurement

The term *code coverage* in software testing denotes the amount of program code which is exercised during the execution of a set of test cases on the system under test. This indicator may simply be used as an overall *coverage percentage*, a proxy for test completeness, but typically more detailed data is also available about individual program elements or test cases. Code coverage measurement is the basis of several software testing and quality assurance practices including white-box testing [66], test suite reduction [80], or fault localization [33].

2.1.1 Different Types and Levels of Code Coverage

Code coverage *criteria* are often used as goals to be achieved in white-box testing: test cases are to be designed until the required coverage level has been reached according to the selected criteria. However, many possible ways exist to define these criteria. They include various granularity levels of the analysis (such as component, method, or statement) and different types of "exercised parts of program code" (for instance, individual instructions, blocks, control paths, data paths, etc). The term *code coverage* without further specification usually refers to statement level analysis and denotes *statement coverage*. Statement coverage shows which instructions of the program are executed during the tests and which are not touched. Even at this level, there may be differences in what constitutes an instruction, which complicates the uniform interpretation of the results. In Java, for instance, a single source code statement is implemented with a sequence of bytecode instructions, and the mapping between these two levels is not always straightforward due to various reasons such as compiler optimization.

Another common coverage criterion is *decision coverage*, where the question is whether both outcomes of a decision (such as an *if* statement) are tested, or if a loop is tested with entering and skipping the body. Since this level of analysis deals with not only individual instructions but control flow as well, coverage measurement at this level imposes more issues.

For instance, Li et al. [53] showed that decision coverage at statement level for the Java language is prone to differences between the source code and the bytecode measurements; they found that practically the two results were hardly comparable. The main reasons for this besides the different optimizations were the actual shortcuts built in the implementation of the logical expressions: what seems to be a single logical expression in the source code can be a very complex control structure in the bytecode.

These examples show that even within a specific language difficulties might occur in defining and interpreting code coverage criteria. This may be even more emphasized at more sophisticated levels such as control path or data-flow based coverages [63].

Coarser granularity level coverage criteria (such as methods, classes, or components) are also common, for instance, when the system size and complexity do not enable a fine-grained analysis. Also, often it is more useful to start the coverage analysis in a "top-down" fashion by starting from the components that are not executed at all, extend the tests to cover that component at least once, and then continue the analysis with lower levels. In particular, *procedure level coverage* is a good compromise between analysis precision and the ability to handle big systems.

In our research, we primarily deal with this granularity, that is, we treat procedures (Java methods in particular) as atomic code elements that can be covered. At this level "covered" means that the method has been executed at least once during the tests but we do not care about what instructions, paths, or data have been exercised in particular. Contrary to what would be expected, this granularity level also involves difficulties in the interpretation of code coverage, which was the main motivation for our research. In particular, we found significant differences between different code coverage measurement tools for Java configured for method level analysis.

2.1.2 Applications of Code Coverage and Risks

Uncertainties in code coverage measurement may impose various risks. Here, we overview the notable applications of code coverage measurement and how they may be impacted by the uncertainties.

The most important application of code coverage measurement is white-box testing (often referred to as *structure-based testing*). It is a dynamic test design technique relying on code coverage to systematically verify the amount of tests needed to achieve a completeness goal, a coverage criterion. This goal is sometimes expected to be a complete (that is, 100%) coverage, however in practice, this high level is rarely attainable due to various reasons. As white-box testing directly uses coverage data, it is obvious that inaccuracies in the coverage results directly influence the testing activity. On the one hand, a small difference of one or two percentages in the overall coverage value is usually irrelevant if that value is used to assess the completeness. On the other hand, an item inaccurately reported to be covered provides false confidence in the code during a detailed evaluation, and it may result in unnecessary testing costs if an item is falsely reported to be uncovered.

Other applications of code coverage measurement include general software quality assessment[1], automatic test case generation [75, 21], code coverage-based fault localization [40, 109], test selection and prioritization [64, 28, 93], mutation testing [92, 39], and in general, program and test comprehension with traceability analysis [72]. As in the case of white-box

[1]`https://www.sonarsource.com/products/sonarqube/`

testing, the inaccuracy of code coverage measurement may affect these activities in different ways.

Certain applications do not suffer that much if the coverage data is not precise. This includes overall quality assessment, where the coverage ratio is typically used as part of a more complex set of metrics for software assessment. Here, a difference of a few percentages usually does not affect the overall score. Program comprehension (and general project traceability) is supported by knowing which program code is executed by which test case. Depending on the usage scenario of this information, inaccurate results may lead to either false decisions or simply an increased effort to interpret the data.

The other mentioned applications have high significance in academic research, and the accuracy and validity of the published results may be affected by the issues with the code coverage data. In coverage-driven test case generation, for instance, the generation engine can be confused by an imprecise coverage tool because a falsely reported non-coverage will keep the generation algorithm trying to generate test cases for the program element.

As another example, in code coverage-based fault localization the program elements are ranked according to how suspicious they are to contain the fault based on test case coverage and pass/fail status. Wrong coverage data may influence the fault localization process because if the faulty element is erroneously reported as not covered by a failing test case, the suspicion will move to other (possibly non-defective) program elements.

2.1.3 Code Coverage Measurement for Java

Java itself is a popular language, and due to its language and runtime design, it can be more easily handled as the subject of code coverage measurement than other languages directly compiled for native code (like C++).

In addition, the increased demand for code coverage measurement in agile projects – where continuous integration requires the constant monitoring of the code quality and regression testing – has led to the appearance of a large set of tools for this purpose, many of which are free of charge and open source. However, it seems that the working principles, benefits, drawbacks, and any associated risks with these tools are not well understood by practitioners and researchers yet.

In Java, two conceptually different approaches are used for coverage measurement. In both approaches, the system under test and/or the runtime engine is *instrumented*, meaning that "measurement probes" are placed within the system at specific points, which enables the collection of runtime data but do not alter the behavior of the system. The first approach is to instrument the *source code*, which means that the original code is modified by inserting the probes, then this version is built and executed during testing. The second method is to instrument the compiled version of the system, *i.e.,* the *bytecode*. Here, two further approaches exist. First, the probes may be inserted right after the build, which effectively produces modified versions of the bytecode files. Second, the instrumentation may take place during runtime upon loading a class for execution. In the following, we will refer to these two approaches as *offline* and *online* bytecode instrumentation, respectively. Some example tools for the three approaches are Clover[2] (source code), Cobertura[2] (offline bytecode) and JaCoCo[1] (online bytecode).

[2]`http://cobertura.github.io/cobertura/`

Different possible features are available in tools employing these approaches, and they also have various benefits and drawbacks. In Table 2.1, we overview the most important differences. Of course, many of these aspects depend on the application context; here, we list our subjective assessment. One benefit of bytecode instrumentation is that it does not require the source code, thus it can be used *e.g.,* on third party code as well. On the other hand, it is dependent on the bytecode version and the Java VM, thus it is not as universal as source code instrumentation. In turn, implementing bytecode instrumentation is usually easier than inserting proper and syntactically correct measurement probe elements in the source code. Source code instrumentation requires a separate build for the instrumented sources, while bytecode instrumentation uses the compiled class files. However, the latter requires the modification of the VM in the online version. Source code instrumentation also allows full control over what is instrumented, while bytecode instrumentation is usually class-based (whole classes are instrumented at once). Online bytecode instrumentation will not affect compile time, but its runtime overhead includes not only the extra code execution time, but (usually a one-time per class load) instrumentation costs too. Finally, the bytecode based results are sometimes difficult to be tracked back to source code, while source code instrumentation results are directly assigned to the parts of the source code [106, 58].

Table 2.1: *Code Coverage Approaches for Java*

Property	Source code	Offline bytecode	Online bytecode
Source code	Needed	Not needed	Not needed
Special runtime	Not needed	Not needed	Needed
Bytecode and VM	Not dependent	Dependent	Dependent
Filtering control	Complete	Partial	Partial
Separate build	Yes	No	No
Results in source	Yes	Partially	Partially
Compile time	Impacted	Impacted	Not impacted
Runtime	Impacted	Impacted	Highly impacted
Implementation	Difficult	Easy	Easy

These numerous benefits of bytecode instrumentation (*e.g.,* easier implementation, no need for source code and separate build) are so attractive that tools employing this technique are far more popular than source code instrumentation-based tools [106]. Furthermore, most users do not take the trouble to investigate the drawbacks of this approach and the potential impact on their task at hand. Interestingly, scientific literature is also very poor in this respect, namely systematically investigating the negative effects of bytecode instrumentation on the presentation of results in source code (see Section 3.2).

The important benefits of source code instrumentation might overweight bytecode instrumentation in some situations, which are visible from Table 2.1. The most important benefit is that in the situations when the results are to be investigated on the source code level (in most of the cases!), mapping needs to be done from the computations made on the bytecode level. Due to the fact that perfect one-to-one mapping is generally not possible, this might impose various risks.

2.2 Fault Localization

Debugging and related activities are among the most difficult and time-consuming ones in software development [112]. It needs to be performed in all phases of the lifecycle when a defect is found, including initial development, during testing, when failures occur in a live system, and when dealing with regression errors during evolution. Debugging activities may easily range up to 50-75% of the total development cost [112]. This activity involves human participation to a large degree, and many of its sub-task are difficult to automate.

A relevant debugging sub-task is *fault localization* (FL), in which the root causes of an observed failure are sought. Fault localization is notoriously difficult, and any (semi)automated method, which can help the developers and testers in this task, is welcome.

2.2.1 Spectrum-Based Fault Localization

There exist a class of approaches to aid FL which are popular among researchers, but have not yet been widely adopted by the industry: Spectrum-Based Fault Localization (SBFL), also known as Statistical Fault Localization (SFL) [97, 69, 89, 71, 77].

The basic intuition behind SBFL is that code elements (statements, blocks, paths, functions, etc.) exercised by comparably more failing test cases than passing ones are considered as “suspicious” (*i.e.,* likely to contain a fault), while non-suspicious elements are traversed mostly by passing tests. Suspiciousness can be expressed in different ways, usually assigning one value to each code element (called the *suspiciousness score*), which can then be used to *rank* the code elements. The idea is that by inspecting this list a developer would find the fault near the beginning of the list, hence being more productive in localizing the fault.

The use of execution profiles – program spectra – for FL purposes has been proposed for the first time in a study on the Y2K problem aimed to discover date-dependent computations [77], although the first mention of the idea appeared already in 1987 [15]. SBFL approaches are using detailed code coverage data *i.e.,* the program spectra and test results to approximate the location of faults with different special suspiciousness formulae.

Since the early years of fault localization, SBFL methods emerged to be one of the main approaches to fault localization [96], and they are still finding their way to be employed in practice [48, 90, 47, 2]. For instance, most studies are carried out using artificial faults [71], and still the faulty element is usually placed far from the top of the rankings [102, 70]. Abreu et al. [1] investigate the accuracy of SBFL approaches in practice. Le et al. [48] show that there is a gap between theoretical and practical results.

Different types of program spectra have been proposed by Harrold et al. [33, 32] (hit-based, count-based, counting branches, paths, dependencies, etc.), however, the most commonly adopted approach uses individual statements or functions as the basic program elements. Researchers proposed many different scoring mechanisms, but these are essentially all based on four fundamental statistics: counts of passing/failing and traversing/non-traversing test cases in different combinations [97, 71]. Popular suspiciousness scores are Tarantula [40], Ochiai [2], and DStar [99], among others. A detailed analysis of suspiciousness formulae is reported in the survey by Wong et al. [97], and in the study of Pearson et al. [71]. Xie et al. [103] examined the equivalence and hierarchy between a number of formulae, while Yoo et al. [111] showed that there does not exist a perfect scoring formula which outperforms known techniques found by humans or even by automatic search-based methods.

2.2.2 Limitations of SBFL Approaches

Recent studies highlighted some barriers to the wide adoption of the SBFL methods, including a high number of suggested elements to investigate [102, 70], the applicability of theoretical results in practice [48], little experimental results with real faults [71], validity issues of empirical research [90], and so on. Kochhar et al. [47] performed a systematic analysis of practitioners' expectations in the field.

One reason why an SBFL formula may fail is what is referred to as *coincidental correctness* [94, 60, 9]. This is the situation when a test case traverses a faulty element without failing. This can happen quite often since not all exercised elements may have an impact on the computation performed by a test case [61], and if there are relatively more such cases than traversing and failing ones, the suspiciousness score will be negatively affected [60].

A further problem is that there are no guarantees that any scoring mechanism will show a sufficiently good correlation between the score and the actual faults [97, 71, 103, 111]. One additional reason an SBFL-based method may fail is that these approaches provide only the ranked list of code elements, however, this gives little or no information about the context of bugs which makes their comprehension a cumbersome task for developers.

2.2.3 Improvements to SBFL Approaches

One of the main reasons for the suboptimal performance of SBFL, in general, is coincidental correctness. This has been the focus of several works [61, 95, 7]. Wang et al. [95] used context patterns for common fault types, which can strengthen the correlations between program failures and the coverage of faulty program entities. Bandyopadhyay and Ghosh proposed an approach to assign weights to test cases for representing their importance in FL based on the proximity to the failing test cases [7]. They also proposed an approach to iteratively predict and remove coincidentally correct test cases based on user feedback in small programs [8].

Xie et al. [103] present an informative overview of suspiciousness score assignment approaches. Yoo [108] presents an automatic approach to derive risk evaluation formulae using genetic programming. Renieris and Reiss [76] use nearest neighbor measures of program spectra for FL. Several researchers have used the *learning to rank* model [52], to combine different FL algorithms [105, 6, 117].

Gong et al. [25] propose to complement SBFL techniques with the users' feedback, showing how this could produce significant improvements on the FL accuracy. Finally, to support developers in visualizing the output of spectrum-based FL, Orso et al. [65] developed Gammatella, a tool that visualizes statement suspiciousness using color maps.

2.2.4 Non-SBFL Approaches

Other than statistical analysis of dynamic test case executions, there have been other approaches proposed for FL as well. These include slicing-based [116]; statistics-based [56, 59]; and mutation-based approaches [68, 67]. Machine learning and data mining techniques are employed for FL as well [98, 13]. Researchers have also proposed model-based FL approaches [62, 100], as well as state-based approaches [113, 115]. We refer to the surveys of Wong et al. [97] and Wong and Debroy [96] and Parmar and Patel [69].

A recent paper by Zou et al. [117] presents an empirical comparison of different FL families of algorithms that include SBFL, mutation-based approaches, program slicing, stack trace analysis, predicate switching, and history-based approaches. They also combine these mechanisms using learning to rank [52].

Other debugging techniques involve record-replay and user interaction [3, 46, 57, 79].

Finally, some debugging approaches are loosely related to fault localization, for example the works of Zeller and Hildebrandt [113], Zeller [112], and Kiss et al. [45] on Delta Debugging, as well as on algorithmic debugging and testing [22, 86]. Delta debugging systematically narrows down failure-inducing circumstances in order to isolate failure causes automatically. In algorithmic debugging, the tester feedback is used to reduce the search space of a faulty program element, but it requires a large execution tree to be constructed. These approaches construct an execution tree, which is pruned based on the user feedback about questions related to particular nodes of the tree.

2.2.5 Evaulation of Fault Localization

Several strategies have been proposed in the literature for measuring the effectiveness of SBFL methods, but they are practically all based on looking at the rank position of the actual faulty element within the list of all possible program elements. One strategy is to express this as the number of elements that need to be investigated by the programmer before finding the fault [97], and another is the opposite: elements that need not to be investigated [76]. This is usually expressed in relative terms compared to the length of the rank list (program size). However, Parnin and Orso [70] argued that absolute rankings are more helpful in practical situations.

Another issue with these mechanisms is the handling of ties [104] because in many cases different program elements may get assigned the same suspiciousness scores. Some approaches select the first (best case), last (worst case) or middle (expected case) element for expressing this value, while others simply treat the elements with the same values as all belonging to one position.

For measuring the effectiveness of fault localization, the strategy to look at "elements that need to be investigated" using the "expected case" in the case of ties is implemented in this dissertation. This is reported in a set of measures called *Expense*, with two variants: an absolute one expressed in the number of code elements (E) and a relative version compared to the length of the rank list (E'). The following formulae express precisely how to calculate these values (following [1]):

$$E = \frac{|\{i|s_i > s_f\}| + |\{i|s_i \geq s_f\}| + 1}{2}\,, \; E' = \frac{E}{N} \cdot 100\%\,,$$

where N is the number of code elements, for $i \in \{1, \ldots, N\}$ s_i is the suspiciousness score of the ith code element and f is the index of the faulty code element.

Recent user studies report that developers tend to investigate only the top 5 or at most the top 10 elements in the recommendation list provided by localization methods before giving up [102, 47]. Hence, any improved rank position which is beyond these thresholds will probably be less useful, no matter how much relative improvement they can achieve. Therefore, we define the notion of *enabling improvement*, as an improvement in which the traditional FL algorithm ranks the faulty program element at a position larger than 10 (or

5), but the evaluated method reaches the faulty element in less than 10 (or 5) steps. This way from a practically "hopeless" localization scenario the method enables the user to localize the fault by inspecting the top elements in the list (the *accuracy* measure by Sohn and Yoo [88] is similar). In this dissertation, the following concrete cases will be reported to express enabling improvement:

- $(10, \infty] \to (5, 10]$ The base FL score is larger than 10 and the method under evaluation reaches the fault in 5 to 10 steps.
- $(10, \infty] \to [1, 5]$ The base FL score is larger than 10 and the method under evaluation reaches the fault within 5 steps.
- $(5, 10] \to [1, 5]$ The base FL score is between 5 and 10 and the method under evaluation reaches the fault fault within 5 steps.

In addition, in some analyses, we also report $(10, \infty] \to [1, 10]$ that is a combination of the first two cases from above.

Part I

Code Coverage Measurement

3 Effects of Measurement Methods on Java Code Coverage and Their Impact on Applications

3.1 Introduction

In software development and evolution, different processes are used to keep the required quality level of the software, while the requirements and the code are constantly changing. Several activities aiding these processes require reliable measurement of *code coverage*, a test completeness measure. As with any other test completeness measure, it does not necessarily have a direct relationship to fault detection rate [36], however, code coverage is widely used and relied upon in several applications. This includes white-box test design, regression testing, selective retesting, efficient fault detection, fault localization, and debugging, as well as maintaining the efficiency and effectiveness of the test assets in a long term [73]. Essentially, code coverage indicates which code parts are exercised during the execution of a set of test cases on the system under test. The knowledge about the (non-)covered elements will underpin various decisions during these testing activities, so any inaccuracies in the measured data might be critical.

Software testers have long established the theory and practice of code coverage measurement: various types of coverage criteria like statement, branch, and others [12], as well as technical solutions including various kinds of instrumentation methods [106]. This work was motivated by our experience in using code coverage measurement tools for the Java programming language. Even in a relatively simple setting (a method-level analysis of medium size software with popular and stable tools), we found significant differences in the outputs of different tools applied for the same task. The differences in the computed coverages might have serious impacts on different applications, such as false confidence in white-box testing, difficulties in coverage-driven test case generation, and inefficient test prioritization, just to name a few.

Various reasons might exist for such differences and surely there are certain issues that tool builders have to face, but we have found that in the Java environment, the most notable issue is how *code instrumentation* is done. The code instrumentation technique is used to place "probes" into the program, which will be activated upon runtime to collect the necessary information about code coverage. In Java, there are two fundamentally different instrumentation approaches: *source code* level and *bytecode* level. Both approaches have benefits and drawbacks, but many researchers and practitioners prefer to use bytecode instrumentation due to its various technical benefits [106]. However, in most cases the application of code coverage is on the source code, hence it is worthwhile to investigate and compare the two approaches. In earlier work [c3], we investigated these two types of code coverage measurement approaches via two representative tools on a set of open-source Java programs. We found that there were many deviations in the raw coverage results due to the various technical and conceptual differences in the instrumentation methods. In this work, we have fine-tuned our measurements based on the previous results, examined and described the deviations in the coverage in more detail, and performed experiments and quantitative analysis on the effect of the differences. Similar studies exist in relation to branches and statements [53].

Extending the earlier experiment, this work reports on an empirical study to compare the code coverage results provided by tools using the different instrumentation types for Java coverage measurement on the method level. We initially considered a relatively large set of candidate tools referenced in literature and used by practitioners, and then we started the experiments with five popular tools which seemed mature enough and actively used and developed. Overall coverage results are compared using these tools, but eventually, we selected one representative for each instrumentation approach to perform the in-depth analysis of the differences (JaCoCo[1] and Clover[2]). The measurements are made on a set of 8 benchmark programs from the open-source domain which are actively developed real-size systems with large test suites. The differences are systematically investigated both quantitatively (how much the outputs differ) and qualitatively (what the causes for the differences are). Not only do we compare the coverages directly, but investigate the impact on a possible application of coverage measurement in more detail as well. The chosen applications are test prioritization and test suite reduction based on code coverage information.

We believe that the two selected tools are good representatives of the two approaches and being the most widely used ones, many would benefit from our results. A big initial question was, however, if we could use the tools as the "ground truth" in the comparison. Since most of the applications of code coverage operate on the *source code*, the source code instrumentation tool Clover was the candidate for this role. Thus, we performed a manual verification of the code coverage results provided by this tool by randomly selecting the outputs for investigation while maintaining a good overall functional coverage of the subject systems. We interpreted the results in terms of the actual test executions and program behavior on the level of source code. During this verification, we did not find any issues, which made it possible to use this tool as a ground truth for source code coverage results.

To perform the actual comparison of the tools, various technical modifications had to be done on the tools and the measurement environment; for instance, to be able to perform per-test case measurements and calculate not only overall coverage ratios. This enabled a more detailed investigation of the possible causes for the difference.

Our results indicate that the differences between the coverage measurements can vary

[1]`http://eclemma.org/jacoco/`
[2]`https://www.atlassian.com/software/clover/`

in a large range and that it is difficult to predict in what situations will be the risk of measurement inaccuracy higher for a particular application. In summary, we make the following contributions:

1. The majority of earlier work on the topic dealt with lower-level analyses such as statements and branches. Instead, we performed experiments on the granularity of Java methods in real-size Java systems with realistic test suites. We found that – contrary to our preliminary expectations – even at this level there might be significant differences between bytecode instrumentation and source code instrumentation approaches. Method level granularity is often the viable solution due to the large system size. Furthermore, if we can demonstrate the weaknesses of the tools at this level, they are expected to be present at the lower levels of granularity as well.

2. We found that the overall coverage differences between the tools can vary in both directions, and in the case of seven out of the eight subject programs they are at most 1.5%. However, for the last program, we measured an extremely large difference of 40% (this was then attributed to the different handling of generated code).

3. We looked at more detailed differences as well with respect to individual test cases and program elements. In many applications of code coverage (in debugging, for instance) subtle differences at this level may lead to serious confusion. We measured differences of up to 14% between the individual test cases, and differences of over 20% between the methods. In a different analysis of the results, we found that a substantial portion of the methods in the subjects was affected by this inaccuracy (up to 30% of the methods in one of the subject programs).

4. We systematically investigated the reasons for the differences and found that some of them were tool-specific, while the others would be attributed to the instrumentation approach. This list of reasons may be used as a guideline for the users of coverage tools on how to avoid or workaround the issues when a bytecode instrumentation-based approach is used.

5. We also measured the effect of the differences on the application of code coverage to test prioritization. We found that the prioritized lists produced by the tools differed significantly (with correlations below 0.5), which means that the impact of the inaccuracies might be significant. We think that this low correlation is a great risk: in other words, it is not possible to predict the potential amplification of a given coverage inaccuracy in a particular application. This also affects any related research which is based on bytecode instrumentation coverage measurement to a large extent.

3.2 Related Work

There is a large body of literature on comparing various software analysis tools, for instance, code smell detection [19], static analysis [18], and test automation [74], just to name a few.

Most of the works that compare bytecode and source code instrumentation techniques focus on the usability, the operability, and the features of certain tools, *e.g.,* [106, 58], but the accuracy of the results they provide is less often investigated – despite the importance and the possible risks overviewed above.

Li et al. [53] examined the difference between source code and bytecode instrumentation considering branches and statements. They concluded that due to several differences between the two methods, source code instrumentation is more appropriate for branch coverage computation. We verify the differences on coarser granularity (on method level), and how these differences impact the results of the further applications of the coverage.

Kajo-Mece and Tartari [43] evaluated two coverage tools (source code and bytecode instrumentation-based ones) on small programs and concluded that the source code-based one was more reliable for use in determining the quality of their tests. We also used Java source code and bytecode instrumentation tools in our experiments but on a much bigger data set in a more comprehensive analysis.

Alemerien and Magel [4] experimented to investigate how the results of code coverage tools are consistent in terms of line, statement, branch, and method coverage. They compared the tools using the overall coverage as the base metric. Their findings show that branch and method coverage metrics are significantly different, but the statement and line coverage metrics are only slightly different. They also found that program size significantly affected the effectiveness of code coverage tools with large programs. They did not evaluate the impact of the difference on the applications of code coverage. We investigated only method-level coverages, but we did not only use overall coverage but analyzed detailed coverage information as well. Namely, we computed coverage information for each test case and method individually and analyzed the differences using this data.

Kessis et al. [44] presented a paper in which they investigated the usability of coverage analysis from a practical point of view. They conducted an empirical study on a large Java middleware application and found that although some of the coverage measurement tools were not mature enough to handle large-scale programs properly, using the adequate measurement policies would radically decrease the cost of coverage analysis, and together with different test techniques, it could ensure better software quality. Although we are not examining the coverage tools themselves, we rely on their produced results and cannot exclude all of their features from the experiments.

This work is a follow-up to our previous work on the topic [c3], in which we investigated bytecode and source code coverage measurement on the same Java systems we used in this work. We found that there were many deviations in the raw coverage results due to the various technical and conceptual differences between the instrumentation methods, but we did not investigate the reasons in detail and how these differences could influence the applications where coverage data was used.

There are many code coverage measurement tools for Java (*e.g.,* Semantic Designs' Test Coverage[3], Cobertura[2], EMMA[4], FERRARI [11], and others). In Section 3.4.2, we discuss how we selected the tools for our measurements.

In this work, we consider test suite reduction and test prioritization, as the application of code coverage. Yoo and Harman [107] conducted a survey on different test suite reduction and prioritization methods among which coverage-based methods can also be found. The most basic coverage-based prioritization methods, which were studied by Rothermel et al. [81], are the *stmt-total* and *stmt-addtl* coverages. In our experiments, we applied these concepts on the method level and referred to them as *general*, *additional*, and *additional with resets*. One of the test prioritization algorithms we used in our experiments was optimized for fault

[3]`http://www.semdesigns.com/Products/TestCoverage`

[4]`http://emma.sourceforge.net/`

localization, which is based on the previous work by Vidács et al. [93]. Fault localization aware test suite reduction turned out to produce different results than fault detection aware reduction, which optimizes code coverage.

3.3 Research Goals

Following earlier research on the drawbacks of bytecode instrumentation for Java code coverage on lower granularity levels [53], and addressing challenges listed in the previous sections, the aim of our research is the following. We investigate in quantitative and qualitative terms in what situations and to what extent Java method-level code coverage based on bytecode instrumentation is different from coverage based on source code. We investigate the differences between the actual coverage information on a detailed level and determine the root causes of these differences after a manual investigation of the source code of the affected methods. In addition, we evaluate the impact of the inaccuracies on an application of code coverage measurement, namely test prioritization, and test reduction. We assume that a certain degree of the differences in the coverages may be reflected in a different degree of inaccuracies of the application.

To achieve our goal, we consider several candidate tools and then conduct an empirical study involving two representative tools, one with source code instrumentation and one with online bytecode instrumentation. We then measure the code coverage results on a set of benchmark programs and elaborate on the possible causes and impacts.

More precisely, our research questions are:

RQ3.1 How big is the difference between the code coverage obtained by an unmodified bytecode-instrumentation-based tool and a source-code-instrumentation-based tool on the benchmark programs?

RQ3.2 What are the typical causes for the differences?

RQ3.3 Can we eliminate tool-specific differences, and if we can, how big is the difference, – which can be possibly attributed to the differences in the fundamental approach, that is, bytecode vs. source code instrumentation – that remains?

RQ3.4 How big is the impact of code coverage inaccuracies on the application in test prioritization, and test suite reduction?

In this work, we calculate and analyze coverage results on the *method level.* More precisely, the basic elements of coverage information are whether a specific Java method is invoked by the tests or not, regardless of what statements or branches are taken in that method. At first, this might seem too coarse a granularity, but we believe that the results will be actionable due to the following.

In many realistic scenarios, coverage analysis is done hierarchically starting from the higher-level code components like classes and methods. If the coverage result is wrong at this level, it will be wrong at the lower levels too. Also, in the case of different applications, unreliable results at the method level will probably mean similar (if not worse) results at the level of statements or branches as well. Previous works have shown that notable differences exist between the detailed results of bytecode and source code coverage measurements at the statement and branch level [53] and that at the method and branch level the overall coverage values show significant differences [4]. So, this leaves the question of whether there are notable differences in method-level coverage results as well open.

3.4 Description of the Experiment and Initial Measurements

To answer the research questions outlined in the previous section, we conducted an empirical study on eight open-source systems (introduced in Section 3.4.1) with code coverage tools for Java employing both instrumentation approaches. Initially, we involved more tools, but as Section 3.4.2 discusses, we continued the measurements with two representative tools. In Section 3.4.3, we overview the measurement process and discuss some technical adjustments we performed on the tools and subjects.

Apart from the coverage measurement tools, our measurement framework consisted of some additional utility tools. The main tool we relied on was the SoDA framework [91]. For the representation of the coverage data in SoDA, the data generated in different forms by the coverage tools were converted into the common SoDA representation, the coverage matrix. Later, this representation was used to perform additional analyses. This framework also contains tools to calculate statistics, produce graphical results, etc. SoDA includes the implementation of the test case prioritization and the test suite reduction algorithms, which we used for our Research Question 3.4. Apart from this, only general helping shell scripts and spreadsheet editors have been used.

3.4.1 Benchmark Programs

For setting up our set of benchmark programs, we followed these criteria. As we wanted to compare bytecode and source code instrumentation, the source code had to be available. Hence, we used open-source projects, which also enables the replication of our experiments. We used the Maven infrastructure in which the code coverage measurement tools easily integrate, so the projects needed to be compilable with this framework. Finally, the subject programs needed to have a usable set of test cases of realistic size, which are based on the JUnit framework[5] (preferably version 4). The reason for the last restriction was that the use of this framework was the most straightforward for measuring per-test case method coverage.

We searched for candidate projects on GitHub[6] preferring those that had been used in the experiments of previous works. We ended up with eight subject programs which belonged to different domains and were non-trivial in size (see Table 3.1). The proportion of the tests in these systems as well as their overall coverage is varying, which makes our benchmark even more diversified. Columns “All Tests” and “Excluded Tests” show the size of the test suites and the number of test cases that were excluded (we discuss the technical modifications that we performed in Section 3.4.3 in detail).

[5]`http://junit.org/`
[6]`https://github.com/`

Table 3.1: *Subject Programs. Metrics were Calculated from the Source Code (Generated Code was Excluded)*

Program	Version	LOC	Methods	All Tests	Excluded Tests	Domain
Checkstyle[a]	6.11.1	114K	2 655	1 589	104	static analysis
Lang[b]	#00fafe77	69K	2 796	3 683	358	java library
Math[c]	#2aa4681c	177K	7 167	5 842	902	java library
Time[d]	2.9	85K	3 898	4 177	162	java library
MapDB[e]	1.0.8	53K	1 608	1 786	68	database
Netty[f]	4.0.29	140K	8 230	4 066	247	networking
OrientDB[g]	2.0.10	229K	13 118	950	153	database
Oryx[h]	1.1.0	31K	1 562	208	0	mach. learning

[a]`https://github.com/checkstyle/checkstyle/tree/checkstyle-6.11.1`
[b]`https://github.com/apache/commons-lang/commit/00fafe77`
[c]`https://github.com/apache/commons-math/commit/2aa4681c`
[d]`https://github.com/JodaOrg/joda-time/releases/tag/v2.9`
[e]`https://github.com/jankotek/mapdb/releases/tag/mapdb-1.0.8`
[f]`https://github.com/netty/netty/releases/tag/netty-4.0.29.Final`
[g]`https://github.com/orientechnologies/orientdb/releases/tag/2.0.10`
[h]`https://github.com/OryxProject/oryx/releases/tag/oryx-1.1.0`

3.4.2 Selection of Coverage Tools

Our goal in this research was to compare the code coverage results produced by tools employing the two instrumentation approaches. Hence, we wanted to make sure that the tools selected for the analysis are good representatives of the instrumentation methods and that our results are less sensitive to tool specificities. The comprehensive list of tools we initially found as candidates for our experimentation is presented in Table 3.2.

We ended up with this initial list after reading the related works (some of them are mentioned in Section 3.2) and searching for code coverage tools on the internet. We learned that the area of code coverage measurement for Java was most intensively investigated in the early 2000s. At that time there were several different tools available, but the support for most of these tools has long ended. There were tools referred by related literature and some webpages which we could not even find, so we did not include them in the table.

In the next step, this list was reduced to five tools, which are shown in the first five rows of the table and marked in boldface. For making this shortlist, we established the following criteria. First, we aimed to actively developed and maintained tools that were popular among users. We measured the popularity of the tool candidates by reviewing technical papers, and open-source projects, and utilizing our experiences from previous projects. The tools had to handle older and current Java versions including new language constructs (support for at least Java 1.7 but preferably 1.8 was needed). Finally, we wanted the tool to easily integrate into the Maven build system[7], as this is one of the popular build systems used in many open-source projects. In addition, the ability of smooth integration reduces the chances of unwanted changes in the behavior of the system and the tests used in the experiments. Finally, among the more technical requirements for the tools was the ability to perform

[7]`https://maven.apache.org/`

Table 3.2: *Tools for Java Code Coverage Measurement*

Tool	**Approach**	**Supported Java/JRE Version**	**Active**	**Licence**
Clover	source	1.3+	present	commercial/free
Cobertura	bytecode	1.5–1.7	2015	free
JaCoCo	bytecode	1.5+	present	free
Jcov	bytecode	1.0+	present	free
SD Test Coverage tools	source	1.1+	present	commercial
Agitar(One)	bytecode	1.6+	present	commercial
CodeCover	source	1.5–1.7	2014	free
Coverlipse	bytecode	1.5	2009	free
EclEmma (JaCoCo-based)	bytecode	1.5+	present	free
Ecobertura (Cobertura-based)	bytecode	1.5–1.7	2010	free
Emma	bytecode	1.5	2005	free
Gretel (by Univ. of Oregon)	bytecode	1.3+	2003	free
GroboUtils	bytecode	1.4	2004	free
Hansel (Gretel-based)	bytecode	1.5	2006	free
InsECTJ	bytecode	1.5	2003	free
Jcover	both	1.2 – 1.4	2009	commercial
Jtest (by Parasoft)	bytecode	?	present	commercial
JVMDI	bytecode	1.4+	2002	free
Koalog	bytecode	?	2004	commercial
NetBeans Code Coverage Plugin	bytecode	1.6	2010	free
NoUnit	bytecode	1.5	2003	free
PurifyPlus	bytecode	1.5+	present	commercial
Quilt	bytecode	1.4	2003	free
TestWorks	bytecode	1.2+	present	commercial

coverage measurement on a per-test basis.

We ended up with five tools to be used for our measurements meeting these criteria. Three of them use bytecode instrumentation, and two are based on source code instrumentation. In Section 2.1.3, we discussed three fundamental code coverage calculation approaches for Java. However, in the case of bytecode instrumentation, there are no fundamental differences in how and which program elements are instrumented, only the "timing" of the instrumentation is different. Hence, we include source code instrumentation as one category, but we do not consider both types of bytecode instrumentation separately in the following, where we discuss the selected tools briefly

3.4.2.1 Source-Code-Based Instrumentation Tools

As mentioned earlier, there are comparably much fewer coverage tools employing this method. Essentially, we could find only two active tools that are mature enough and meet our other criteria to serve the purposes of our experiment. The tools selected for the source code instrumentation approach were Clover by Atlassian[2] (version 4.0.6), and Test Coverage[8] by Semantic Designs (version 1.1.32).

Clover is the product of Atlassian, and it was a commercial product for a long time but, eventually, it became open source. It handles Java 8 constructs, easily integrates with the Maven build system, and can measure per-test coverage.

Test Coverage is a commercial coverage tool from Semantic Designs. Native, it works on Windows, handles most Java 8 code, and can be integrated into the Maven build process as an external tool. Per-test coverage measurement is not feasible by this tool, because it could only be solved by the individual execution of test cases.

We performed some initial experiments to compare these two tools. The details and results of this investigation can be found in Section A.1. Results showed that there were only minimal differences in the outputs produced by the two tools, and their accuracy is almost the same.

Finally, we chose Clover to be used in our detailed bytecode-source code measurements because it has better Maven and per-test coverage measurement support, which made it easier to integrate it into our experiments. Also, Test Coverage had difficulties in handling some parts of our code base, which would have required their exclusion from the experimentation.

To be able to use the source code instrumentation results as the baseline in our experiments, we did a manual verification of the results of Clover by performing a selective manual instrumentation. A subset of the methods were selected for each of our subject systems, up to 300 methods per system. Then, we manually instrumented these methods and ran the test suite. We interpreted the results in terms of actual test executions and program behavior on the level of source code. When the results were checked, we found no deviations between the covered methods reported by the manual instrumentation and by Clover. Thus, we treat Clover as a "ground truth" for source code coverage measurement from this point onward.

3.4.2.2 Bytecode-Based Instrumentation Tools

We found three candidate tools in this category that met the mentioned criteria: JaCoCo[1] (version 0.7.5.201505241946), Cobertura[2] (version 2.1.1), and JCov[9] (version c7a7c279c3a6). Contrary to the two previous ones, all three tools in this category are open-source. We performed similar experiments to compare the results of these three tools, and investigated their differences. These experiments and the results are detailed in Section A.2. We concluded that the main cause of the differences was mostly due to the slightly different handling of compiler-generated methods in the bytecode by the three tools (such as for nested classes). Since the quantitative differences were at most 4% and they were considering mostly generated methods, which are less important for code coverage analysis, we concluded that one representative tool of the three should be sufficient for further experiments.

We ended up using JaCoCo[1] for the bytecode instrumentation approach thanks to its popularity and slightly higher visibility and easier integration for use in our experiments than the other two. This is a free Java code coverage library developed by the EclEmma team, which can easily be integrated into a Maven-based build system. JaCoCo has plug-ins for

[8]`http://www.semdesigns.com/Products/TestCoverage/JavaTestCoverage.html`

[9]`https://wiki.openjdk.org/display/CodeTools/jcov/`

most of the popular IDEs *i.e.,* Eclipse[10], NetBeans[11], IntelliJ[12], for CI- and build systems *e.g.,* Jenkins[13], Maven[7], Gradle[14] and also for quality assessment tools *e.g.,* SonarQube[1]. These plug-ins have about 20k installations/downloads per month in total. In addition, several popular projects, *e.g.,* Eclipse Collections[15], Spring Framework[16], and Checkstyle[17] are utilizing JaCoCo actively. JaCoCo has up-to-date releases and an active community.

3.4.3 Measurement Process

To be able to compare the code coverage results and investigate the differences in detail, we had to calculate the coverages with the different settings and variations of the tools. In particular, we wanted the data from the two tools to be comparable to each other, and we wanted to eliminate tool-specific differences. Hence, we essentially calculated different sets of coverage data, which we will denote by JaCoCo^{glob}, JaCoCo, JaCoCo^{sync}, Clover^{glob}, Clover, and Clover^{sync}, with explanations following shortly.

The experiment itself was conducted as follows. First, we modified the build and test systems of each subject program to integrate the necessary tasks for collecting the coverage data using the two coverage tools. This task included a small modification to ignore the test failures of a module that would normally prevent the compilation of the dependent modules and the whole project. This was necessary when some tests of the project failed on the measured version, and in a few cases when the instrumentation itself caused some tests to fail. Furthermore, to avoid any bias induced by “random” tests, we executed each test case three times and excluded from the further analysis the ones that did not yield the same coverage consistently every time. Eventually, we managed to arrive at a set of filtered test cases that were common for both tools. Column “Excluded Tests” of Table 3.1 gives the number of excluded tests for each subject.

Since we planned a detailed study on the differences between the tools, we wanted to make sure that we could gather *per-test case* and *per-method* coverage results from the tools as well (*i.e.,* which test cases covered each method and the opposite). Clover could be easily integrated into the Maven build process and there were no problems in producing the per-test case coverage information we needed. JaCoCo measurements could also be integrated into a Maven-based build system, but originally it could not perform coverage measurements for individual test cases. So, to be able to gather the per-test case coverage information, we implemented a special listener at first. Then, we configured the test execution environment of each program to communicate with this listener. As a result, we were able to detect the start and the end of the execution of a test case (tools and examples are available at[18]). From the per-test case coverage, we then produced a *coverage matrix* for each program, which is essentially a binary matrix with test cases in its rows, methods in the columns, and 1s in the cells if the given method is reached when executing the given test case. From this matrix,

[10]`https://www.eclipse.org/`
[11]`https://netbeans.apache.org/`
[12]`https://www.jetbrains.com/idea/`
[13]`https://jenkins.io/`
[14]`https://gradle.org/`
[15]`https://www.eclipse.org/collections/`
[16]`https://spring.io/projects/spring-framework/`
[17]`https://checkstyle.sourceforge.io/`
[18]`https://github.com/sed-szeged/soda-jacoco-maven-plugin`

we could easily compute different kinds of coverage statistics including per-test case and per-method coverage.

Due to the mentioned extension of the JaCoCo measurements, we essentially started with two different kinds of JaCoCo results: the original one without test case separation, which we will denote by JaCoCo$^{\text{glob}}$, and the one with the special listener denoted by JaCoCo. Theoretically, there should be no differences between the two types of measurements, but since we noticed some, we investigated their amount and causes. Table 3.3 shows the two overall coverage values for each program in columns two and three, with the differences shown in the fourth column. It can be observed that JaCoCo results are always somewhat smaller than the JaCoCo$^{\text{glob}}$ measurements. The difference is caused by executing and covering some general utility functions (such as the preparation of the test execution) in the unseparated version during the overall testing, but these cannot be associated with any of the test cases. Since these methods have no covering test cases assigned, when we summarize the coverage of all test cases, the methods remain uncovered. Note, that Clover does not suffer from this issue as it originally produces per-test case results.

Table 3.3: *Effect of Technical Setup on Overall Coverage Values*

Program	**JaCoCo$^{\text{glob}}$**	**JaCoCo**	**Difference**	**Clover$^{\text{glob}}$**	**Clover**	**Difference**
Checkstyle	53.85%	53.77%	-0.08%	93.82%	93.82%	0.00%
Lang	93.29%	92.92%	-0.37%	93.28%	93.28%	0.00%
Math	85.59%	84.92%	-0.67%	84.65%	84.65%	0.00%
Time	91.36%	89.52%	-1.84%	89.94%	89.94%	0.00%
MapDB	79.65%	74.64%	-5.01%	76.06%	76.06%	0.00%
Netty	47.41%	40.92%	-6.49%	46.66%	40.18%	-6.48%
OrientDB	38.40%	27.01%	-11.39%	39.84%	28.01%	-11.83%
Oryx	29.62%	29.51%	-0.11%	27.51%	28.75%	+1.24%

Another technicality with the Clover tool had to be addressed before moving to the experiments themselves. Namely, for handling multiple modules in projects we had two choices with this tool: either to integrate the measurement on a global level for the whole project or to integrate it individually in the separate sub-modules (this configuration can be performed in the Maven build system). Since JaCoCo follows the second approach, we decided to configure Clover individually for the sub-modules as well. These measurements will be denoted simply by Clover and will be used subsequently.

Three of the eight subject systems (Netty, OrientDB, and Oryx) include more than one sub-module, so this decision affected the measurement in these systems. To assess how much the handling of sub-modules differs from the other approach, we performed global measurements as well (denoted by Clover$^{\text{glob}}$), whose results can be seen in the last three columns of Table 3.3. Clover$^{\text{glob}}$ measurements typically include a smaller number of covered elements than Clover, but the coverage itself can be bigger, which is due to the different number of overall recognized methods. We will elaborate on the differences caused by sub-module handling in detail in Section 3.5.2.

3.5 Results

The experiment results presented in this section follow our RQs from Section 3.3. As discussed in Sections 2.1.3 and 3.4.2, we treat source-code-based instrumentation as more suitable for source code applications and Clover results as the ground truth, hence JaCoCo results will be compared to Clover, serving as the reference.

3.5.1 Differences in Unmodified Coverage Values

Our first research question dealt with the number of differences we can observe in the overall coverage values calculated by the two tools. In this phase, we wanted to compare the raw, unmodified data from the tools “off the shelf”, because this could reflect the situations users would experience in reality as well. However, as explained in Section 3.4.3, we needed to perform a modification of the tool execution environment to enable per-test case measurements, which caused slight changes in the overall coverages. In this section, we rely on this modified set of measurements, which is denoted simply by JaCoCo and Clover.

3.5.1.1 Total Coverage

First, we compared the overall method-level coverage values obtained for our subject programs, which are shown in Table 3.4. JaCoCo and Clover results are shown for each program, along with the difference in the coverage percentages. Coverage ratios are given in percentages of the number of covered methods from all methods recognized by the corresponding tool.

Table 3.4: *Overall Coverage Values for the Unmodified Tools*

Program	JaCoCo	Clover	Difference
Checkstyle	53.77%	93.82%	-40.05%
Lang	92.92%	93.28%	-0.36%
Math	84.92%	84.65%	+0.27%
Time	89.52%	89.94%	-0.42%
MapDB	74.64%	76.06%	-1.42%
Netty	40.92%	40.18%	+0.74%
OrientDB	27.01%	28.01%	-1.00%
Oryx	29.51%	28.75%	+0.76%
Average	**61.65%**	**66.84%**	**-5.19%**

Excluding the outlier program Checkstyle, the differences between the tools range in a relatively small interval, from -1.42% to 0.76%. In the following sections, we seek the reasons for the differences, and we will explain the outlier in Section 3.5.2.4.

3.5.1.2 Per-Test Case Coverage

While Table 3.4 presents the overall coverage values produced by the whole test suite, the coverage ratios attained by the individual test cases might show another range of specific differences. Table 3.5a contains statistics about the coverages for the individual test cases for JaCoCo, and Table 3.5b shows similar results for Clover. (Coverage is again the number of covered methods relative to all methods). This includes minimum, maximum, median, and average values with standard deviation. In Table 3.5c the difference in the average values between the two tools is shown (positive values denote bigger average coverage values for Clover). It can be observed that Checkstyle reflects the high global difference between Clover and JaCoCo in the per-test case results too, although not as emphasized as in the global case. Interestingly, in the case of MapDB, Netty, and Oryx the average individual differences between Clover and JaCoCo have the opposite sign than the global differences.

Note, that it is not obvious how individual coverage differences imply global coverage differences and vice versa. Individual coverages could differ greatly but the overall coverage is not changed. For example, one test case is enough for a method to be reported as covered, and if one instrumentation technique reports a hundred covering test cases while the other technique reports only one, the global coverage will not change only the individual ones. Similarly, low average individual differences might result in a high global difference; if many test cases have only one method which is reported differently, and these methods are uniquely covered by those test cases, the small individual differences will sum up in a high global coverage difference.

An even more detailed way to compare the per-test case coverages is by investigating not only the overall coverage ratios but the whole *coverage vector*, *i.e.,* the row vector of binary values from the coverage matrix for the corresponding test case. Figure 3.1 shows the analysis of the difference between the corresponding coverage vectors produced by the two tools. The difference was computed as the Hamming distance normalized to the vector lengths. Note, that the two tools may recognize a different number of methods (more on this in the next section), so in these cases, the vectors were padded with no-coverage marks for the missing methods. Then, the distribution of the obtained differences was calculated and shown as a histogram. The x and y axes of the graphs show the ranges of differences (size ranges are 1%), and the number of cases (relative to all cases) for the given difference range respectively. As expected, a lot of small differences occurred. In particular, a significant portion of the vectors had 0 differences. On the other end, none of the programs had vectors with Hamming distance values larger than 20%. Hence, to ease readability, we omit the values that were 0 or larger than 20% from the diagrams and show the corresponding numbers instead in the top right corner of the graphs.

There are two interesting results here. The Hamming distances of MapDB have a different distribution than the other programs (the differences go up to 14% with this program), and not surprisingly, it is also reflected in the higher average per-test case differences. This shows that while differences would occur in either direction, in most of the cases the JaCoCo coverage turns out to be higher than the Clover result (in contrast to the others, where on average Clover reports higher coverage). The second interesting observation is that Checkstyle behaves differently than the other programs: the high average per-test case difference measured for this program (1.64%) is not observable from the Hamming distances (more than 90% of the test cases show no difference and the others are below 1%). This seems to be inconsistent at the first sight. However, as we will explain in Section 3.5.2.4, Checkstyle shows a significant difference in the number of methods detected by the two instrumentation techniques. Thus, the average coverage values for JaCoCo and Clover used significantly different denominators, while during Hamming distance computation a common denominator

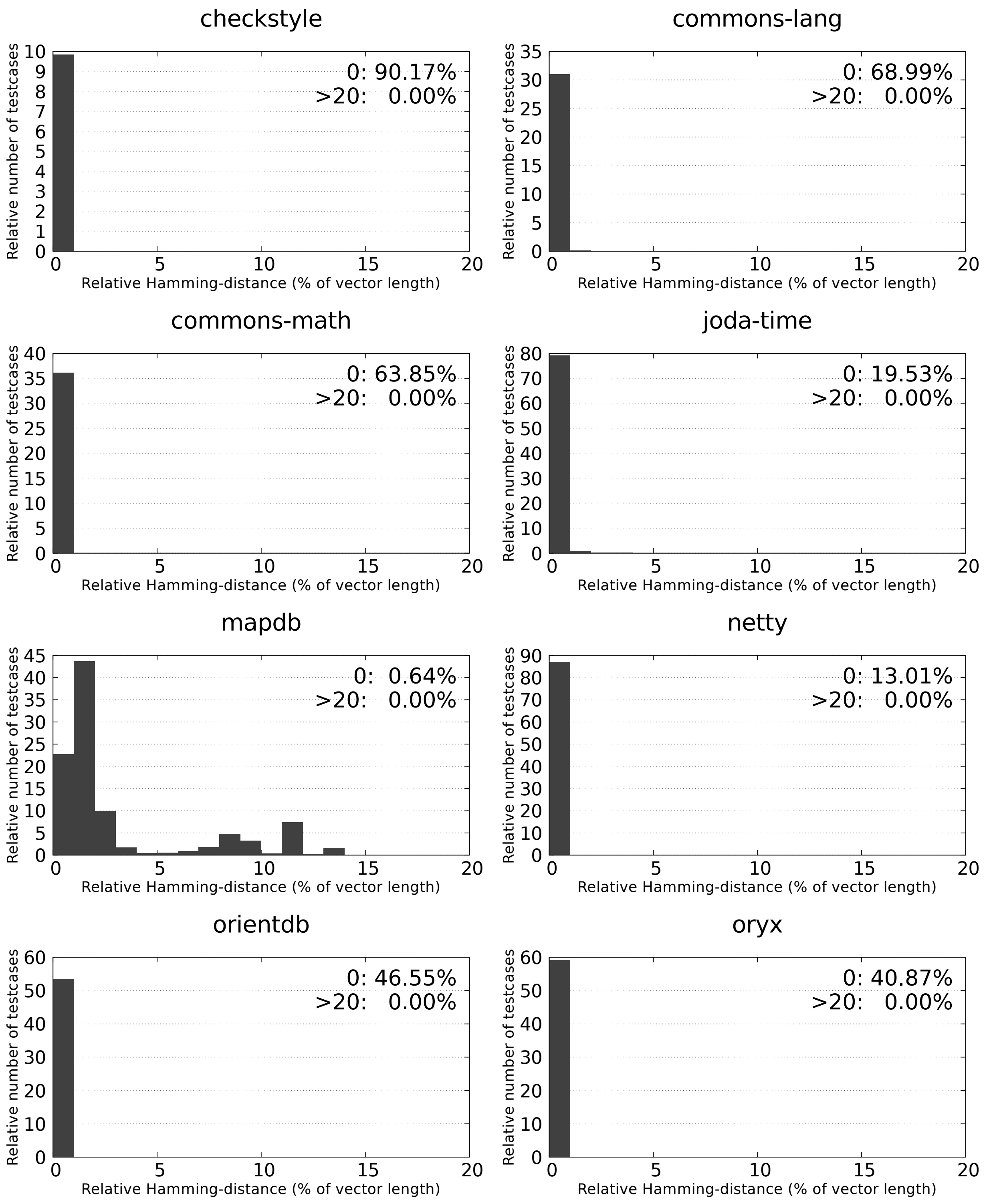


Figure 3.1: *Relative Hamming distances of test case vectors (JaCoCo vs. Clover)*

Table 3.5: *Per-Test Case Coverages*

(a) *JaCoCo Results*

Program	Minimum	Maximum	Median	Average	Deviation
Checkstyle	0.00%	15.87%	4.11%	3.02%	2.36%
Lang	0.00%	3.10%	0.20%	0.61%	0.74%
Math	0.00%	4.34%	0.26%	0.47%	0.54%
Time	0.00%	8.84%	1.24%	1.62%	1.37%
MapDB	0.00%	20.86%	6.28%	7.67%	5.39%
Netty	0.00%	3.43%	0.24%	0.32%	0.29%
OrientDB	0.00%	9.40%	0.22%	0.58%	1.20%
Oryx	0.00%	2.05%	0.33%	0.45%	0.40%

(b) *Clover Results*

Program	Minimum	Maximum	Median	Average	Deviation
Checkstyle	0.00%	30.13%	6.21%	4.66%	3.65%
Lang	0.00%	2.93%	0.21%	0.64%	0.76%
Math	0.00%	4.34%	0.25%	0.47%	0.55%
Time	0.00%	9.93%	1.23%	1.64%	1.39%
MapDB	0.00%	22.08%	6.09%	7.19%	6.08%
Netty	0.00%	3.69%	0.26%	0.37%	0.33%
OrientDB	0.00%	9.88%	0.24%	0.62%	1.28%
Oryx	0.00%	1.79%	0.38%	0.48%	0.40%

(c) *JaCoCo to Clover Average Difference*

Program	Average Difference
Checkstyle	+1.64%
Lang	+0.03%
Math	0.00%
Time	+0.02%
MapDB	-0.48%
Netty	+0.05%
OrientDB	+0.04%
Oryx	+0.03%

was used and this caused the observed difference.

3.5.1.3 Per-Method Coverage

In the previous experiment, we investigated the coverage from the test case dimension. In this one, we did the same from the method dimension. The distributions of the Hamming distances were calculated similarly to the per-test case analysis. The results in Figure 3.2 show a similar overall picture to the per test-case analysis. Therefore, we used the same method to exclude and emphasize the differences larger than 20% or equal to 0. The distribution of the distances and the average per-test case coverage values seem to be unrelated. However, Checkstyle and MapDB behave differently than the other programs in this case, too. The high average per-test case difference measured for Checkstyle is not observable from the Hamming distances, while the high distances in the case of MapDB result in a relatively high average difference.

In this case, we performed another, slightly different analysis. For each method, we recorded how many of the test cases cover that method according to the two tools. Then, we counted the number of the methods for which the number of the covering test cases was equal, and how many times one or the other tool reported this differently. This kind of an analysis is useful because it helps to find out the number of situations when the methods are found falsely (not) covered, which may lead to confusion in certain applications.

When we compared the "number of covering test cases", we identified three kinds of differences. First, JaCoCo and Clover recognized different sets of methods, for which the reasons will be explained in Section 3.5.2.4. Second, for some of the methods recognized by both approaches, Clover reported at least one covering test case but JaCoCo did not, and vice versa. The third kind of difference is when both tools reported that a method was covered, but by a different number of test cases. Figure 3.3 shows the associated results. In particular, the percentage of the methods is shown for each program (with respect to the total number of methods recognized by any of the tools) for the following cases: there is no difference in the covering sets of test cases, and either Clover or JaCoCo reports more covering test cases. In the latter category, all three kinds of differences from above are counted together.

An ideal case would be if only *Equals* is present, which would mean that the two tools completely agree on the coverages. However, we can observe that the situation is quite different. First, many methods are not recognized by the Clover tool, which can be attributed to various reasons but mostly to the generated code. A notable outlier is Checkstyle with 55% of such methods, the others are below 15%.

Next, as can be seen in Table 3.6, there are only a few methods for which Clover and JaCoCo do not agree in the coverage fact (covered by at least one test case) while both recognize the method ("Czero" and "Jzero" columns). We investigated all these 220 methods manually to find out the reasons for the difference (see Section 3.5.2.4). The other two columns report on the cases when the number of covering test cases was not zero but different. Column "CltJ" means "Clover reports less than JaCoCo", while "JltC" is "JaCoCo reports less than Clover". A significant portion of the methods in the subjects was affected by the inaccuracy to some extent (nearly 30% for MapDB and over 11% for Time).

Answer to RQ3.1: The detailed per-test case and per-method measurements, when compared to the overall coverage ratios, may show quite different trends. In some cases, the overall ratios are reflected in the detailed data, but not necessarily: a high overall difference is often caused by a little difference on a detailed level, and the opposite. In other words, by observing a certain overall level of inaccuracies, we cannot predict the differences on the more detailed levels, and consequently, the effect on possible applications.

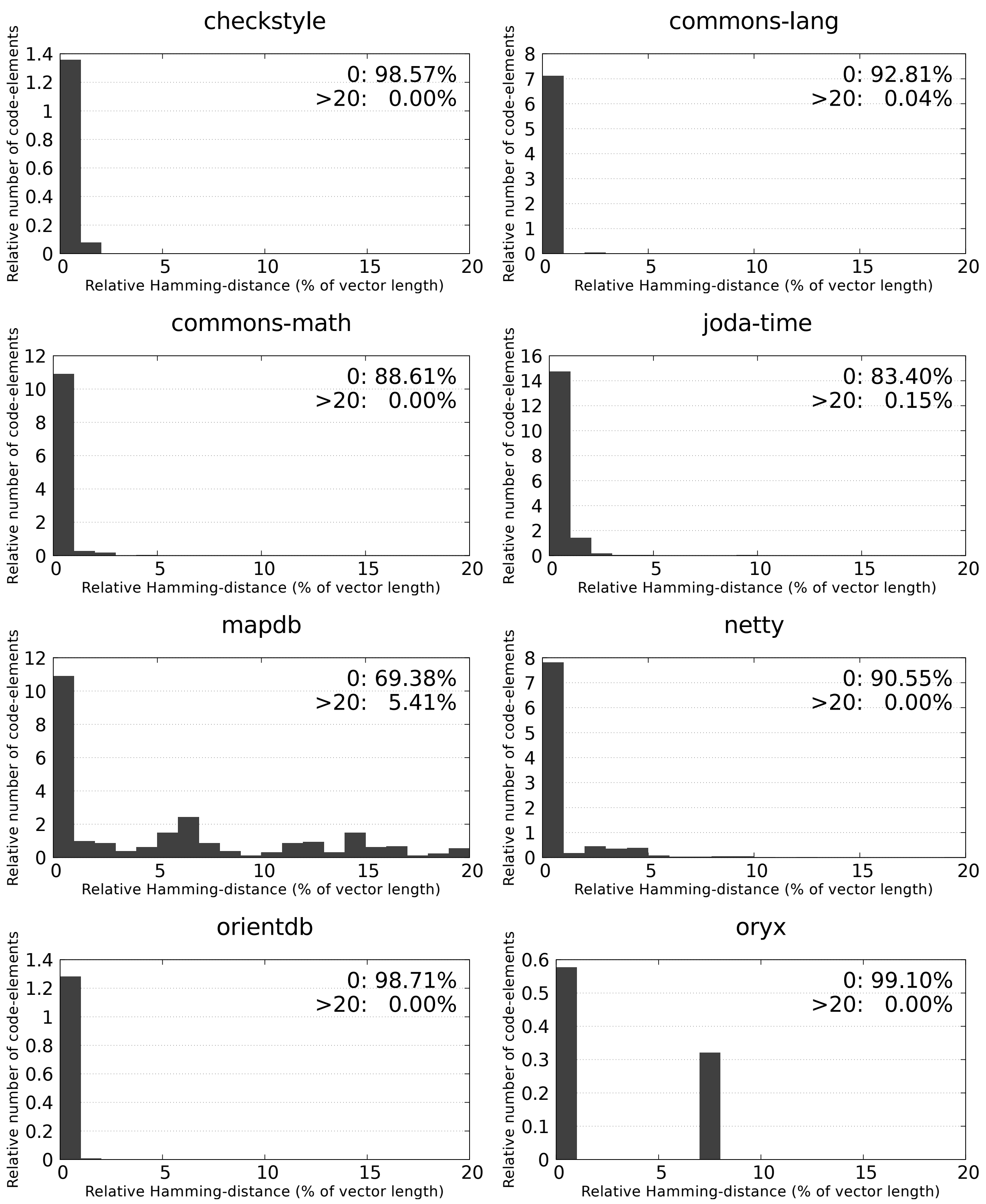
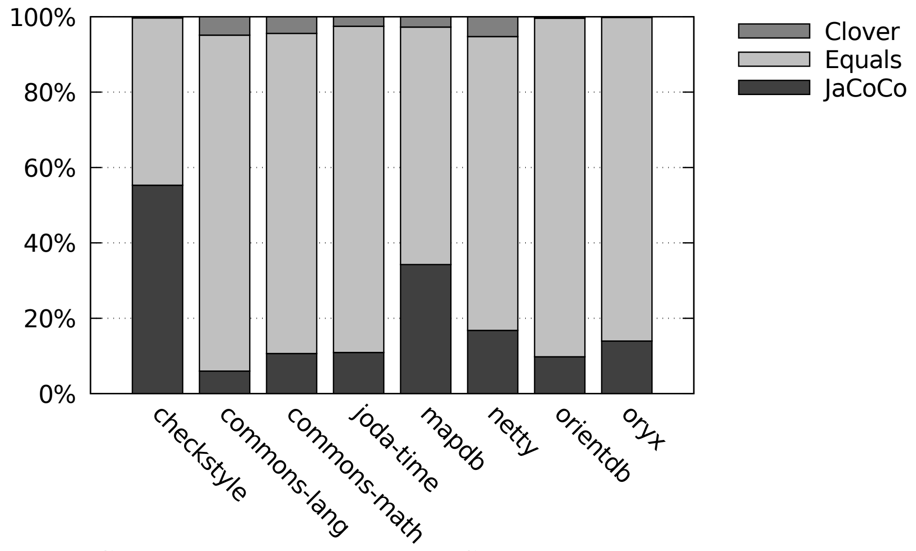


Figure 3.2: *Relative Hamming distances of code-element vectors (JaCoCo vs. Clover)*



Clover: Clover reports more covering TCs; **Equals:** both tools report the same covering TCs; **JaCoCo:** JaCoCo reports more covering TCs.

Figure 3.3: *Summary of differences in the per-method coverage*

Table 3.6: *Differences in Per-Method Coverages of Code Elements of JaCoCo and Clover*

Program	Czero	CltJ	JltC	Jzero
Checkstyle	1	9	16	0
Lang	0	21	131	5
Math	19	297	239	7
Time	0	358	86	2
MapDB	7	450	25	2
Netty	91	300	466	76
OrientDB	1	104	32	5
Oryx	4	8	1	0

3.5.2 Causes of Differences

In this section, we address the possible causes for the differences we observed and presented in the previous section. We used manual inspection, and carefully examined the differences between the coverage results reported by JaCoCo and Clover. Due to their large number, we could not look into each difference, instead, we manually selected the typical cases making sure that each system and module was sufficiently covered by our investigation. We also made sure to investigate all of the most problematic cases shown in columns “Czero” and “Jzero” of Table 3.6. We involved the original and instrumented versions of the source code and the bytecode as well. In addition, we examined other artifacts like build configuration files to reveal additional factors that could be the cause of differences. The work was performed by three authors of the paper, first by dividing the cases equally and performing the inspection individually. Then, each result was cross-validated by at least one of the other authors to ensure consistency of the results. Altogether, we manually investigated several hundred individual methods and test cases one by one during this work. Finally, we were able to identify a set of common reasons, which we overview in the following.

3.5.2.1 Cross-Sub-Module Coverage

In the case of projects consisting of multiple sub-modules, Clover and JaCoCo work differently. Clover first instruments the whole source, thus, it can report cross-module coverage. On the other hand, JaCoCo concentrates on the tested module and does not instrument other modules when it is tested, thus it cannot report a cross-module coverage. Consider Figure 3.4 for illustration. Let the system have three sub-modules A, B, and C, which define their dependencies and build processes including unit tests. In the example, modules A and C include test cases, while module B does not. The arrows on the figure indicate the possible calls from tests to non-test methods and between non-test methods. During the build (and test phase) of module C, module A is treated as an "external" dependency, which prevents JaCoCo from instrumenting and measuring the coverage of the methods of A (along the gray edges starting from module C). Thus, it only considers a method of module A covered if the method is invoked from the tests of module A (along the black edges). On the other hand, Clover aggregates the coverage among all modules, so if a method from A is used in a test in C (through some gray edges), Clover considers the method as covered. These different behaviors can lead to differences in the global coverage of the projects. Our subjects Netty, OrientDB, and Oryx are examples of multiple-module projects. The other five programs are single-module projects.

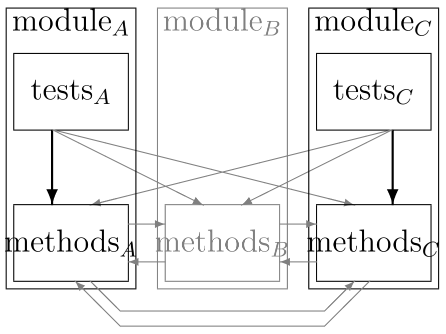


Figure 3.4: *Illustration of problems with sub-modules*

Note, that although we investigated only Maven-based projects in our experiments, we think that similar problems may occur in other build configuration systems as well.

3.5.2.2 Untested Sub-Modules

In the case of JaCoCo, if a module does not have any tests its methods will not be recognized. Consider again Figure 3.4, where module B does not have any tests, thus JaCoCo will not be executed for it (grayed in the figure). Consequently, the methods of B will not be recognized and they will be missed from the set of all methods of the project. Clover, on the other hand, correctly determines the set of all methods across all sub-modules and will include methods of modules A, B, and C.

3.5.2.3 Test Case Preparation and Cleanup

Some test cases might need preparation or cleanup, and this is common in some programs. Technically, this is usually implemented as *setup* and *teardown* methods (annotated by `@Before`, `@BeforeClass`, `@After`, or `@AfterClass` in JUnit) associated with a test class or a set of test methods. These are executed before/after a set of test cases or before/after each test case that requires them. In the JaCoCo measurement architecture, these are counted as part of the test cases, *i.e.,* all the methods executed during these setup/teardown phases are reported as covered by the corresponding test cases. On the contrary, Clover does not treat setup and teardown as an integral part of the test case, and as a consequence, if a method is covered only during the setup or teardown phase of a test case, it will not be assigned to the test case.

3.5.2.4 Recognized Method Sets

In addition to the previous three cases, a further inaccuracy exists between the JaCoCo and Clover results regarding the method sets because the set of methods detected from the source code and the bytecode can differ. There are many reasons for this; some of them are the inherent problems of the measurement and some of them are tool-specific. Table 3.7 introduces our measurements in this regard (see also Figure 3.3). The second column shows how many methods are recognized by both tools, and how many are recognized only by Clover or JaCoCo, which are given in the third and fourth columns, respectively. The last column contains the sum of these three values, *i.e.,* the total number of methods recognized by Clover and JaCoCo together.

Table 3.7: *Number of All Methods*

Program	Both	Clover Only	JaCoCo Only	Total
Checkstyle	2 653	2	3 263	5 918
Lang	2 783	13	154	2 950
Math	7 080	87	221	7 388
Time	3 884	14	76	3 974
MapDB	1 585	23	150	1 758
Netty	8 195	35	1 297	9 527
OrientDB	13 097	21	1 306	14 424
Oryx	1 560	2	244	1 806

Observe that several methods are recognized only by Clover or JaCoCo. The second group is not surprising because we expected in advance a relatively large number of generated methods in the bytecode (due to the necessary mechanisms of the Java language, on which we will elaborate shortly). However, we were somewhat surprised to see that some methods were recognized only by Clover. This section also investigates the reasons for this difference. In any case, the impact of the difference in the recognized method sets can be significant. The results from Section 3.5.1 were all produced for the two tools which were based on a different total number of methods. This makes difficult, for instance, the comparison of the overall coverage ratios because they involve different denominators for the two tools. The actual causes of the different method sets are overviewed below.

Test methods Unit tests themselves should not be investigated for coverage, hence all methods of unit test classes needed to be excluded from further analysis. JaCoCo relies on the project description to determine the test methods. On the other hand, Clover tries to determine test methods by checking the class and method names, and in most cases this is reliable. However, in some cases when the test class names did not follow the naming conventions, Clover misclassified the tests as regular methods.

Compiler-generated code The difference in favor of JaCoCo consisted of various methods generated by the Java compiler, *e.g.,* default constructors (if they were not given in the source), `<clinit>` methods, and access methods in the case of some nested class operations. Generated methods are considered for the coverage analysis by most bytecode instrumentation tools – including JaCoCo, however, a source-code-based tool like Clover may not include them. This issue results in additional methods appearing in bytecode coverage results, which can increase or decrease the overall coverage value.

Generated code All programs we investigated included code constructs that result in compiler-generated methods, which are not visible in source code, only in bytecode (*e.g.,* default constructors and initializers). On the other hand, some projects generate a portion of the *source code* of the application on-the-fly using some external tools like ANTLR, or a configuration setting. In particular, most of the big differences between JaCoCo and Clover results of Checkstyle were caused by this reason (see Table 3.4). The two tools handle this kind of code differently: while JaCoCo includes them in the same manner as any other regular code, Clover excludes them from the analysis. Since the tests of Checkstyle do not cover any of the generated code, the result is that JaCoCo uses a larger denominator than the other tool with a similar amount of covered elements in the nominators. In general, instrumentation tools may handle this situation differently, but usually, they can be configured to consider the generated section of the source code as part of the code base.

3.5.2.5 Instrumentation

We found that in some cases the instrumentation itself modified the behavior of the tests, which might have influenced the list of executed methods. An example is in the Time program, where two specific test cases failed after being instrumented by Clover. This is because the tests utilize Java reflection to query the number of nested classes of the tested class, and – as Clover implements coverage measurements and test case detection by inserting nested classes into the examined class – these two tests failed on assertions right at the beginning of the test case. Similar failures occurred in the Checkstyle project as well, where two of the test cases check whether the classes have a fixed number of fields. However, with the additional fields that Clover inserts in the classes, these assertions fail.

3.5.2.6 Exception Handling During Coverage Measurement

When JaCoCo instruments the bytecode, it inserts probes into strategic locations by analyzing the control flow of all methods of a class. If the control flow is interrupted by an exception between two probes, JaCoCo will not consider the instructions between the probes to be covered. The reason is that if a method throws an exception at the beginning of the caller method, JaCoCo marks the caller method as not covered because it misses the instrumentation probe on the exit point of the method. However, the instrumentation strategy of Clover can handle this situation and it will mark the caller method as covered because it simply considers the probe at the entry point of the method. Another instance of this issue was that JaCoCo computes lower coverage for tests that are expected to throw some exceptions (*i.e.,* annotated as `@Test(expected=SomeException)`. It is related to the above-mentioned exception handling, and it is a known issue of JaCoCo.

3.5.2.7 Name Encoding

A common reason for the differences was related to enums, anonymous and nested classes. The problem is that in some cases a method of such a class may get additional parameters when compiled to bytecode to access the members of its enclosing class. In other cases, the methods even lost some of their source code parameters. This resulted in different signatures of the source code and bytecode instances of the same method.

For example, a constructor like `MyEnum(String name)` of an `enum` type in the `pack` package will have the signature `pack/MyEnum/MyEnum(LString;)V` in the source code, while the bytecode-based tools will see it as `pack/MyEnum/MyEnum(LString;ILString;)V` because of technical requirements. Another example is when there is a private static class named `Bar` with a private constructor `Bar(final Foo f)` nested in a final class named `Foo`. The source-code-based tools recognize the constructor as `Foo$Bar(LFoo;)V`, while bytecode-based ones will see `Foo$Bar()V`.

Such missing or extra parameters in the bytecode make the signatures of these methods different in JaCoCo and Clover measurements. This difference prevented the automatic assignment of the methods of the two measurements and caused the reduction of JaCoCo coverage counts in our experiments.

3.5.2.8 Other

We also found some other, occasional reasons for the deviations. The first one was the different handling of some built-in methods of the `Object` class (for example, `equals`, `finalize`, or `hashcode`). If these were redefined through multiple inheritance levels, both tools occasionally produced incorrect results for these methods. Due to this difference, both JaCoCo and Clover could report lower coverage on the same project. Another reason was that Clover had issues detecting the test cases that were called from test cases (see, for example, the class `c` in the Math project), which resulted in incorrect elements in the coverage results. Although it is possible to avoid calling test cases from test cases (even transitively), if this happens for any reason the resulting detailed coverage data might be unreliable. Note that the overall coverage of the test suite will not be influenced by this issue because the coverage will not be missed just recorded at a different program point.

3.5.2.9 Summary of Difference Causes

During our investigations of the differences listed above, we used the following approach. We tried to eliminate or fix the issues one by one in the hope to reach a state when the measurements produced by the two tools were synchronized. This way, we would have been able to categorize each difference as tool-specific or approach-specific. Unfortunately, we were not able to uniquely classify all difference causes to one of the two categories, as we detail below.

First, we excluded the test cases that were failing because of the instrumentation, but this was rather a workaround than a solution. Second, we eliminated cross-module related issues by measuring these sub-modules individually, and we filtered out those methods from the covered set of a test case that were executed only during the setup/teardown phase. This was appropriate to eliminate certain kinds of differences in our experiments, but in real applications it might eliminate important coverage information (depending on the definition and implementation of a test case and whether module or whole system coverage is needed). In addition, we relied on the Maven project hierarchy and examined the source path information of the classes, and filtered out those methods that were located in the test source directories, *e.g.,* `src/test`. This was required to filter out methods that were incorrectly treated as non-test methods.

To mitigate the remaining inaccuracies, we tried to synchronize the method sets (to make the individual test case and method coverage result comparable), for which we defined a set of criteria. We thought that, as software engineers usually work with source code, the synchronized set should be the set of methods actually appearing in the source code. We wanted to verify if Clover produced this list accurately. For that we used the SourceMeter static analysis tool[19], and found that there were no differences between the two lists in any of the programs. Thus, for each subject program, we created a list of methods based on the source code (excluding, *e.g.,* compiler-generated methods). We also made an assignment between the JaCoCo and Clover methods by hand. With this workaround – although the two tools recognized the same methods with different names – we could compare their coverage values for the individual methods.

We denote the results using these sets as JaCoCo$^{\text{sync}}$ and Clover$^{\text{sync}}$. The important property of the synchronized sets is that they are based on the same set of methods, hence in this step we eliminated inaccuracies in the method sets regardless of their reason.

To summarize, Table 3.8 shows how the issues we found persist in the different measurements. The first column names the issue, the second one states whether we considered the issue being clearly tool-specific, approach-specific, or something in between. We did not categorize any of the issues as purely "approach" specific because we think that any of the potential differences could be theoretically aligned in source code and bytecode instrumentation. However, a number of such issues are not expected to be handled equivalently in a realistic tool or this would be impractical. For example, the standard name encoding in bytecode would be unusual to identically follow in source code (see the enum example above). The third column in the table shows whether the corresponding issue was present in the JaCoCo$^{\text{glob}}$ measurements, and the last two columns show whether the issue caused differences in the two kinds of comparisons we performed. Indeed, we found that most of the specific issues of JaCoCo are present in Cobertura and JCov, the other two bytecode instrumentation tool we considered, as well.

[19]`https://www.sourcemeter.com/`

Table 3.8: *Presence of Issues with Different Levels of Filtering*

Issue	**Tool Specific**	**JaCoCo^{glob}**	**JaCoCo Clover**	**JaCoCo^{sync} Clover^{sync}**
1. Cross-submodule coverage	yes	–	–	–
2. Untested sub-modules	yes	–	–	–
3. Test case preparation and cleanup	yes	●	●	–
4. Recognized method sets	partially	●	●	–
5. Instrumentation	yes	–	○	○
6. Exception handling during measurement	yes	●	●	●
7. Name encoding	partially	●	●	○
8. Other	partially	●	●	●

Measurement (Column 3) –: issue is not present in measurement; ●: issue is present in measurement.
Comparisons (Columns 4–5) –: caused differences can be and are automatically eliminated; ○: caused differences are manually eliminated; ●: differences are present.

Answer to RQ3.2: We were able to identify 8 common reasons, which are detailed in previous sections and summarized in Table 3.8. In addition, we tried to fix these issues, but the results varied: we managed to eliminate most of them, however, some (e.g., exception handling and other technical issues) remained and this work can take significant effort because it might not be fully automated. Also, we found that although there were some tool-specific issues, most of them are generalizable, and will probably be applicable to other bytecode-based and source-code-based tools.

3.5.3 Differences Due to the Instrumentation Approach

In the previous section, we listed the causes of differences in the coverages produced by the two measurement tools. Some of them turned out to be due to tool-specific design decisions, while others seemed to be inherent due to the differences in the fundamental approach, namely bytecode vs. source code instrumentation. In some cases, we could not determine if a specific difference belonged to the "tool-specific" or "approach" category. By using the synchronized method sets and eliminating other tool-specific differences that were possible, we arrive at the JaCoCo^{sync} and Clover^{sync} sets of measurements. The differences in these we attribute to most probably the fundamental differences in bytecode vs. source code instrumentation, however, we cannot be sure that there are no more tool-specific issues present. In this section, we quantitatively compare these two coverages. We take a look again at the total coverage ratios, as well as the per-test case and per-method details.

Table 3.9 shows the comparison of all three aspects at a general level, in which the final results of Section 3.5.1 are repeated for convenience, and the corresponding data are presented for the synchronized versions.

The differences in the overall coverages are shown in columns 2–7 of the table. As expected, in the synchronized set of results there are fewer differences between the two measurements (the largest difference is 0.64%, in contrast to 40.05% of the outlier in the previous set). These results also indicate that the coverage of JaCoCo is never greater than that of Clover with the synchronized set of methods. This suggests that bytecode instrumentation typically demonstrates the *safe but imprecise* case, because a smaller coverage may lead to wasted effort but not to false confidence.

Table 3.9: *Differences in Overall Coverage with the Original and Synchronized Versions of the Tools*

Program	JaCoCo	Clover	Orig. Diff.	JaCoCo^{sync}	Clover^{sync}	Sync. Diff.
Checkstyle	53.77%	93.82%	-40.05%	93.81%	93.81%	0.00%
Lang	92.92%	93.28%	-0.36%	93.12%	93.30%	-0.18%
Math	84.92%	84.65%	+0.27%	85.23%	85.35%	-0.12%
Time	89.52%	89.94%	-0.42%	90.17%	90.19%	-0.02%
MapDB	74.64%	76.06%	-1.42%	76.03%	76.11%	-0.08%
Netty	40.92%	40.18%	+0.74%	39.79%	40.43%	-0.64%
OrientDB	27.01%	28.01%	-1.00%	28.02%	28.05%	-0.03%
Oryx	29.51%	28.75%	+0.76%	28.67%	28.67%	0.00%
Average	**61.65%**	**66.84%**	**-5.19%**	**66.86%**	**66.99%**	**-0.13%**

The comparison of the per-test case results is contained in columns 2–3 of Table 3.10. Here, the overall Hamming distances can be compared, which have been computed jointly for all test cases from the respective coverage matrices. It can be observed that the average differences are reduced in different degrees: while in the case of Lang the reduction was minimal, the difference almost disappeared in the case of Oryx. The reduction is dependent on the internal structure and relations of the programs' methods and tests, and cannot be directly predicted from the different properties we measured in other experiments.

Table 3.10: *Differences in Per-Test Case and Per-Method Coverages with the Original and Synchronized Versions of the Tools*

Program	Orig. H.	Sync. H.	Orig. Strict	Orig. NoStrict	Sync. Strict	Sync. NoStrict
Checkstyle	0.005%	0.002%	25	1	16	0
Lang	0.014%	0.012%	152	5	132	5
Math	0.034%	0.005%	536	26	251	7
Time	0.157%	0.031%	444	2	308	2
MapDB	3.062%	1.165%	475	9	81	3
Netty	0.155%	0.013%	766	167	484	98
OrientDB	0.013%	0.008%	136	6	61	6
Oryx	0.187%	0.004%	9	4	1	0
Average	**0.450%**	**0.160%**	**318**	**28**	**167**	**15**

The per-method coverage differences are also presented for the final results for the respective measurement levels in columns 4–7 of Table 3.10. They show the number of test cases when there is a disagreement between the two tools according to the two levels of strictness, as explained in Section 3.5.1.3. In particular, columns "Orig. Strict" and "Orig. NoStrict" correspond to the sums "CltJ"+"JltC", and "Czero"+"Jzero" from Table 3.6, respectively, while the other two are the same for the synchronized measurements. As can be observed, the synchronization improves this measurement as well, and the improvement rates are again very different for the individual subjects. The counts roughly halved both in the strict and non-strict cases, however, there are notable cases when this was more significant (*e.g.,* MapDB) and also where it was much smaller (*e.g.,* Lang).

Answer to RQ3.3: While we have seen that there might be notable differences in the "synchronized" data sets showing differences due to the instrumentation approach, tool-specific ones cannot always be sorted out reliably.

3.5.4 Impact on Test Case Prioritization and Test Suite Reduction

In Section 2.1.2, we listed leading applications of code coverage measurement, and how they are possibly impacted by the inaccuracies of the tools. The results presented earlier in this section showed that the inaccuracies may directly impact some of the applications, most notably white-box testing, and that this can be directly measured/predicted. However, it does not directly follow if the inaccuracies would have a similar effect in applications where code coverage is indirectly used to achieve a different purpose. We selected code coverage-based test case prioritization and the related test suite reduction [81, 107] to quantify the impact of code coverage inaccuracies.

Informally, *test case prioritization* takes the list of test cases of a test suite and produces a specific order of their execution, which is believed to maximize the chances of early defect detection, localization, and correction. Typically, defect detection is the primary concern, but in this work, we concentrate on both detection and localization. The goal of the former is to have failing test cases because they indicate that there are faults *somewhere* in the system. On the other hand, in fault localization, we aim to find the causes of the faults, in other words, pinpoint to the *location* of actual defects in the code. Both activities may be aided by the use of code coverage information, but coverage needs to be used differently:

- For fault detection, the usual approach is to maximize coverage at the beginning of the prioritized list because it is naturally expected that elements that are not covered by the test cases may not exhibit faults.
- On the other hand, successful localization highly depends on how much the test cases can exhibit different program behavior; *i.e.,* if the test cases show similar behavior on different program elements, these elements may be indistinguishable in this respect. Consequently, for fault localization, those test cases should be chosen that distinguish between different program elements. This is often quite different than simply the highest coverage. Many fault localization algorithms exploit this fact, such as Raptor [27], FLINT [110], and Partition-based [93].

A practical use of the prioritized list of test cases is that not all of them are executed, but only the first N elements of the prioritized list are selected. This can happen in various settings. First, if faults are detected or localized the testing may be terminated. Second, test selection may be terminated at the first point where maximum coverage (or a suitable fault localization metric) is reached. Finally, if there are resources to execute only a fixed number of tests, this is a suitable approach because the chances of successful testing are maximized by using a suitable prioritization algorithm. The test suites are then either minimized by permanently discarding the remaining test cases or limited only for the execution [107]. Section 3.5.4.2 deals with test suite reduction, which is based on the prioritized list of test cases investigated in Section 3.5.4.1.

Our rationale for selecting these applications is that they have a solid algorithmic background and the outcome of the algorithms may significantly influence test effectiveness and efficiency.

In this section, we rely on our first set of coverage data used in Section 3.5.1 for RQ3.1. These are the "raw" coverages produced by the tools and are not influenced by our "synchronization" efforts for RQ3.2 and RQ3.3. We do this because, in most cases, the coverage tools and their results are used "as is": the users do not make an effort to additionally process the data. Thus, we use coverage data denoted by JaCoCo and Clover for these comparisons.

3.5.4.1 Test Case Prioritization

In this experiment, we used four test case prioritization strategies: three optimized for maximal fault detection, and one for fault localization. There are many strategies for coverage-based fault detection prioritization, but the so-called *general* and *additional* are probably the most widely used ones [81]. We will also consider a variation of the second one called *additional with resets.*

The *general* strategy greedily assigns a higher rank to those test cases that produce the highest absolute coverage (that is, the test cases are simply ordered by their coverage value). The *additional* algorithm is a bit more clever in that it looks for test cases that contribute the most to the not yet covered elements (that is, it starts with the highest covering test case and then it greedily selects the test cases based on their additional coverage). A common issue with these approaches is that once high coverage is attained, the algorithm cannot select but randomly chooses from the remaining test cases. Therefore, an extension to the second algorithm (*additional with resets*) restarts the greedy selection once no improvement in the additional coverage can be obtained. In particular, it resets the coverage counter to zero whenever the maximum coverage is reached and continues to append test cases to the prioritized list as if it was creating a new list from the remaining test cases [81].

Our selected algorithm for fault localization aware test case prioritization was the *partition-based* algorithm [93]. The basic idea behind this algorithm is that the code elements in the coverage matrix are *partitioned* according to their coverage patterns produced by the test cases (that is, equal matrix columns constitute a partition). Since the code elements should be distinguishable from each other during fault localization, the finest partitioning is sought in this prioritization. The algorithm greedily selects the test cases that best divide the existing partitions that were obtained with the earlier test cases. To do this, in each step, it divides the code elements into covered and uncovered subsets, and then it is recursively invoked on these subsets to choose the new test cases.

These four algorithms typically produce quite different rankings in the prioritized lists, but all of them are highly sensitive to the coverage data. Some of them mostly depend on the overall coverage (such as *general*), while fine details in the coverage patterns may have a big influence on the others, for example, *partition-based*. Hence, in the first experiment, we wanted to compare each algorithm by how their prioritized lists differ when computed by the two coverage tools.

Note that certain test suite prioritization algorithms (and reduction algorithms as well) make arbitrary choices in cases when more than one item has the same priority value. Therefore, due to their non-deterministic nature, they could produce different results even on the same coverage data. Hence, we designed our algorithms and their inner data structures to be deterministic.

To compare the prioritizations, we used Kendall's τ_B rank correlation coefficient, which is known to be suitable for handling ties.

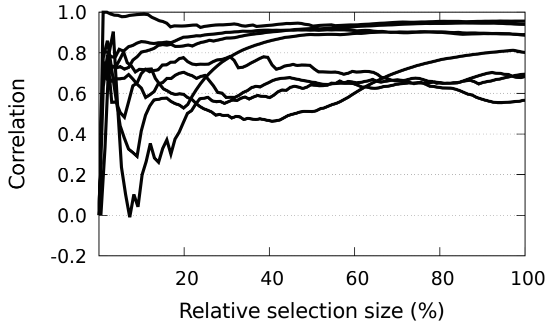
Table 3.11: *Prioritization: Kendall's τ_B Correlation Between JaCoCo and Clover Results*

Program	**General**	**Additional**	**Additional with resets**	**Partition-based**
Checkstyle	0.801	0.274	0.666	0.049
Lang	0.890	0.373	0.751	0.043
Math	0.886	0.267	0.682	0.055
Time	0.940	0.245	0.672	0.018
MapDB	0.695	0.124	0.815	0.064
Netty	0.567	0.129	0.476	0.266
OrientDB	0.956	0.329	0.736	0.364
Oryx	0.684	0.440	0.659	0.379

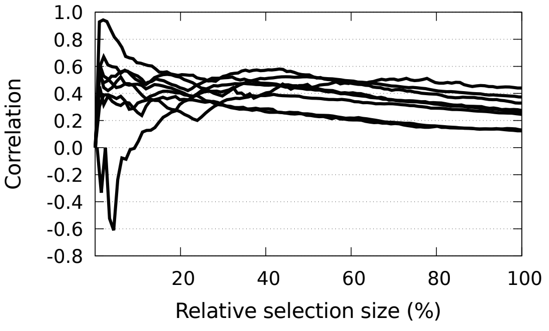
Table 3.11 summarizes the associated results. We can observe that the correlation for the *general* strategy is high or moderate for all programs, leading to the conclusion that the differences in the coverage tools have a relatively low influence on the final rankings produced by this algorithm. The ranked lists produced by the *additional with resets* strategy show a similar correlation, only slightly lower than for the *general* strategy (an exception is subject MapDB). However, the third column shows that the *additional* strategy is much more influenced by the minor differences in the coverage measurements: the highest correlation for this strategy (0.44) is much worse than the lowest one for *general* (0.567), and it has some really low values too (0.124).

These results are caused by the different behavior of the algorithms and how they react to the individual differences of the coverage vectors. In all of them, when there are more test case candidates to be selected as the next item of the prioritized list, the selection is arbitrary (the "random effect"). The good performance of the *general* strategy can be attributed to the fact that before reaching the maximum coverage, the overall coverage is dominant and the smaller differences do not have a big impact here. In the case of this algorithm, the random effect will be smaller, and it will increase only after reaching the maximum coverage. On the other hand, the different behavior of the *additional* and *additional with resets* strategies is more related to the random effect: *additional* reaches maximum coverage relatively early, and then it starts extending the list with elements in an arbitrary order. The *additional with resets* strategy is more deterministic: after the reset, it will work with the same determinism, again and again, hence it will more resemble the behavior of *general*.

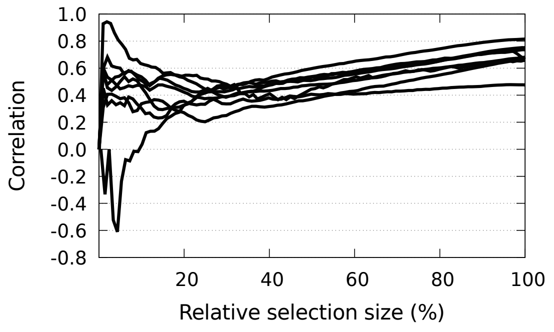
Finally, the *partition-based* strategy for fault localization prioritization is the most sen-



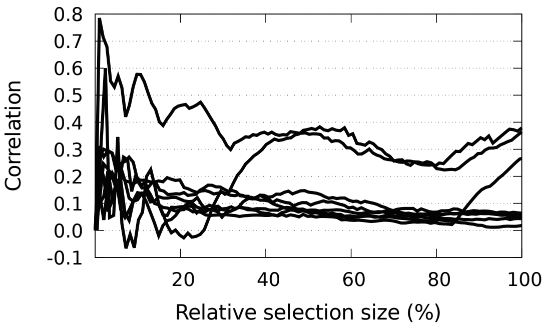
(a) general strategy



(b) additional strategy



(c) additional with resets strategy



(d) partition-based strategy

Figure 3.5: *Test case prioritization: correlation for different selection sizes and strategies*

sitive to coverage differences. Except for maybe OrientDB, Oryx and Netty, it shows no correlation at all. Note that these are the subject programs that have modules, the others are single-module programs. In addition, these programs have low coverage. When selecting the next item in the ranked list, the partition-based strategy prefers the test case that best splits the method sets into two parts. In each step, the algorithm tries to split a partition into parts of mostly equal size, and select the best test case for it. But all the uncovered methods form one partition, which is very large in these cases (due to the low coverage). When this large partition is to be split, the selection of the test case is arbitrary, and the implementation will select the first test case in the original list. Thus, the correlation between the two ranked lists will depend on the correlation of the original lists. This correlation is higher for the modular programs because the lists of the test cases are implicitly grouped by the modules.

The correlation values from the previous table were computed for the whole prioritized lists of test cases. However, often only the beginning of the prioritized list is used (such as for test suite reduction, which is discussed in the next section). Further, the differences in the code coverage measurements can vary depending on the stage of the algorithm: they may be different among the first selected test cases and later when more test cases are already processed. Hence, to see these effects in more detail, we computed the correlation values for each selection size starting from the first test case in the prioritized list up to the whole set.

More precisely, Kendall correlation coefficients were computed for each prefix of the ranked lists, which are shown in Figure 3.5. In these graphs, the x-axis shows how many elements from the beginning of the two prioritized lists were considered for computing the correlation (in percentage), while the y-axis shows the actual correlation value. The curves correspond to the different programs, but we did not distinguish between them because it bears no information for this discussion. The last value in each graph corresponds to the values in Table 3.11. As can be seen, the different strategies behave differently, but it is common that, in the beginning, all strategies are very sensitive to the small coverage differences. Also, the calculation of the correlation is statistically less significant with a smaller number of elements in the data sets. In particular, the corresponding p-values are greater than 0.05 for the first 20–50 elements. Hence, the investigation of the data in these charts should not focus on the very beginning of the curves (also graphically they show quite erratic behavior).

The specific observations we can make from this data are the following. In the case of the *general* strategy (Figure 3.5(a)) the curves are forming a V-shape: the correlation is suddenly reduced and then grows back as we compare more and more elements of the prioritized lists. The characteristics of these drops are program dependent but at some point the correlations grow back and become steady when more elements are considered. This shows that small coverage differences have local influences on the ranking, but globally they have a small impact.

The results for the *additional* strategy (Figure 3.5(b)) are different: as this strategy incorporates some random factor once the test cases in the first part of the prioritized list give full coverage, the correlation after this point starts to decrease as the effect of the arbitrary selection accumulates. However, the curves for the different programs are more similar to each other than those of the general strategy.

Results for the *additional with resets* strategy (Figure 3.5(c)) are very similar to the results of *additional* until the first reset (it is typically between 10 and 30 percent of the test cases). However, due to the elimination of the random factor with the reset, the remaining parts of the curves show similar behavior to the *general* strategy. As more elements are compared, the correlation between the two lists grows until the full lists are compared, and this final correlation is comparable to the *general* strategy.

Since the *additional with resets* strategy is often seen as the best coverage-based greedy strategy for fault detection prioritization, this result indicates a high risk when using different code coverage tools for this application. Namely, when a relatively low number of tests are selected, the influence of the coverage tool is quite high on the results.

Finally, the *partition-based* strategy (Figure 3.5(d)) behaves very differently. As mentioned above, this strategy selects the next test case in the ranking, which best splits the method sets into two parts. As a result, the first very few elements on the two lists for each program are expected to be the same given the relatively small difference in the coverage vectors. But further selections use smaller partitions of the method sets, thus it is more probable that multiple test cases indicate the same “best” split. In this case, the selection of the next one is arbitrary among the candidates. It might happen that two “best” test cases split the corresponding method partition in different ways, which heavily affects the subsequent selections and rankings.

For 5 programs, after the first erratic values of the short list comparisons, the correlation drops down showing that the lists derived from the two coverage measurements are very different; practically, we cannot observe any correlation between them due to the cumulative differences mentioned above. These are single-module programs, where all test cases are potential candidates. However, the modular architecture programs OrientDB, Oryx, and Netty have inherent partitioning aligned with the module boundaries. This means that the number of potential candidates for a selection is limited and randomness has a smaller effect. The *partition-based* strategy results are aligned with these inherent partitions regardless of the coverage measurement method; the algorithm works as if it were ranking the test cases for each module independently. This results in more correlated rankings than for single-module programs. The results for all the programs show that the first 2–3 elements match exactly, the next few are the same but in a different order, and from there onward the lists show very low overall correlation. To conclude, the partition-based prioritization algorithm is very sensitive to the small coverage differences of the individual test cases, which is also true for multi-module programs, but especially holds for single-module ones.

3.5.4.2 Test Suite Reduction

As mentioned, in test suite reduction, a given number of elements from the beginning of the prioritized list is selected. For this experiment, we followed two scenarios. In the first one, we stop the selection when the current subset of the test cases reaches the coverage of the unreduced test suite, and then compare the attained reductions. In the second scenario, we measure the differences in the coverages when any fixed size of the reduced subset is used. These experiments include reductions based on the *general*, *additional*, and *additional with resets* prioritization strategies.

Note, that in both scenarios the measurements for *additional* and *additional with resets* strategies are the same: before the first reset *additional with resets* is equivalent to *additional*, but since the first reset occurs when the maximum possible coverage is reached, the differences in the coverages after this point will be constant. So, we will present *additional* and *additional with resets* results together.

Table 3.12 shows the results for the first scenario: namely, the reduction values for the three fault detection algorithms that could be obtained for the subsets of test cases, which achieve the original coverage of the unreduced sets. The reduction is given as a relative number of eliminated test cases. Apart from the obvious advantage of the *additional* strategies over *general* (which is not the topic of this paper), the differences between the two coverage tools are not as obvious as we have seen for test case prioritization. For the general strategy, MapDB seems to be an outlier; the difference between the reduction rate of the results achieved by the two coverage methods is more than 11%, which could be a consequence of the high Hamming distances between the coverage vectors for this program (see Figures 3.1 and 3.2). On the other hand, Checkstyle and Oryx, for instance, have very different Hamming distances but still, they demonstrate similar test suite reduction differences.

The results of our second reduction strategy are shown in Figure 3.6. Here, we calculated the effects of the differences for various fixed reduction values from 0–100% (in these diagrams, the test suite sizes increase from left to right). We used *general*, *additional*, and *additional with resets* strategies, and calculated the relative differences of the overall coverage values compared to Clover. The x-axis represents the number of test cases (relative to the size of the whole test suite) and the y-axis shows the difference itself. Again, we do not distinguish between the subjects because only the trends are important.

It can be observed that the behavior of different programs in the *general* measurements (Figure 3.6(a)) are different, but in general, the smaller size the reduced suite has, the greater difference can be measured between the coverages of the reduced suites. In other words, a higher reduction rate introduces more uncertainty in the results. In general, we found that

Table 3.12: *Test Suite Reduction without Reducing Coverage for the Different Strategies*

Program	**General**			**Additional Additional with resets**		
	JaCoCo	**Clover**	**Difference**	**JaCoCo**	**Clover**	**Difference**
Checkstyle	0.20%	3.03%	2.83%	75.95%	78.04%	2.09%
Lang	0.69%	0.72%	0.03%	69.95%	70.16%	0.21%
Math	0.08%	0.28%	0.20%	77.04%	77.53%	0.49%
Time	0.54%	0.92%	0.38%	76.01%	76.21%	0.20%
MapDB	0.11%	11.17%	11.06%	89.28%	89.34%	0.06%
Netty	0.23%	0.57%	0.34%	87.85%	88.40%	0.55%
OrientDB	0.25%	0.25%	0.00%	70.63%	71.76%	1.13%
Oryx	7.21%	3.36%	3.85%	59.13%	60.09%	0.96%

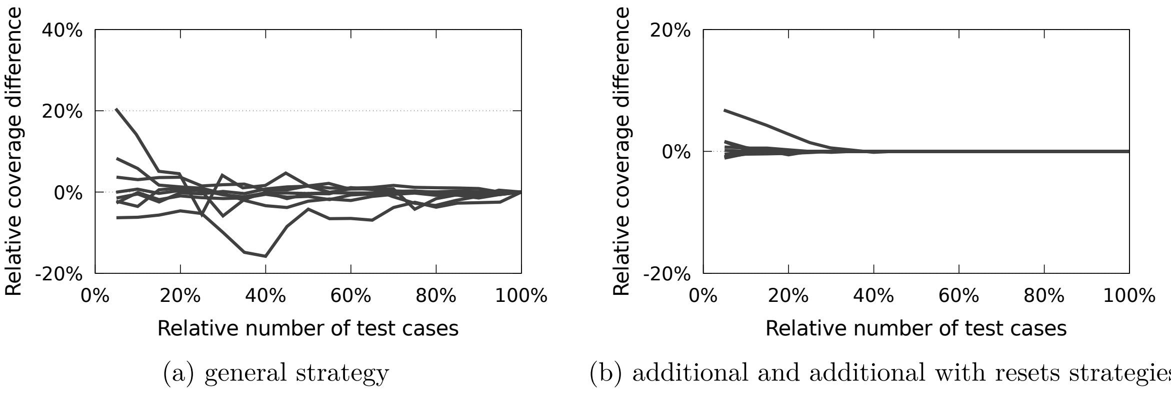


Figure 3.6: *Test suite reduction: coverage differences for general and additional strategies*

if the size of the reduced suite was over 20% of the full suite then the difference in coverage mostly remained under 5%. The shape of the difference depends on the properties of the subject program and its tests. The situation is different for the *additional* measurements (Figure 3.6(b)). The decrease in coverage differences is visible, and it is even much faster. Except for one program, the difference in coverage remains below 2% at a 10% test suite size and above. The fast convergence is caused by the algorithm itself, which aims to reach full coverage as quickly (with as few test cases) as possible. Thus, the two coverage values will approach their maximum with monotonically smaller steps, which implies gradually smaller and smaller differences.

Answer to RQ3.4: Our investigations about the impacts of code coverage inaccuracies showed that they were unpredictable on the chosen applications. For some algorithms, the impacts were high and less for others. A notable example is the test case prioritization algorithm *additional with resets*, which is often considered the best greedy strategy. The influence of the coverage tool was quite high in this case when a relatively low number of tests were selected (which is often the case in practice).

3.6 Discussion

3.6.1 Interpretation of the Results

In this work, we performed experiments on the granularity of Java methods to find out the differences between the bytecode instrumentation approach and source code instrumentation with respect to the final code coverage results. In particular, most of the detailed experiments have been performed using two tools, JaCoCo and Clover, which we selected as representatives of the two instrumentation approaches. We selected Clover as the source-code-based tool from two candidates in our shortlist (which produced very similar results) and used it as the comparison basis in the experiments. We started with three tool candidates in the other category, but we found out that they produce similar results, so we selected JaCoCo as the representative of bytecode-based approaches. In our experiments, we relied on real-size Java systems with realistic test suites, so we believe that testing practitioners and researchers can benefit from our findings as well.

We have seen that the bytecode level and the source code level coverage measurements can produce very different results (answering RQ3.1). In general, the overall differences are low (below 1.5%), but the different properties of the subject systems and the measurement methods may result in very large differences as well. This can be exemplified by the subject Checkstyle, where the generated methods caused a difference of about 40%. Furthermore, differences can be identified in both directions: in some cases, JaCoCo reports more coverage than Clover and vice versa.

On a more detailed analysis level, per-test case and per-method differences also showed discrepancies in both directions. Overall, in some cases the differences are minimal (below 1%), however since this is very much project-dependent, we measured relatively high differences as well (higher than 20% in some cases; see Figures 3.1 and 3.2). The differences might affect a large portion of the methods of a program, even around 30%, as can be observed from Table 3.6.

The causes of these differences are various (RQ3.2). There are tool-specific ones like the different sub-module handling of the used tools, or the handling of the test setup and teardown methods; these are independent of the selected instrumentation method. These can be eliminated by filtering the results (although this might not be fully automated). Other tool-specific features like the influence of the instrumentation on the behavior of the subject system tests are integral parts of the tools and in general, cannot be avoided. Finally, deviations like the different issues on method set recognition and name encoding are mostly determined by the instrumentation method, not the tool.

Theoretically, some of these differences could be eliminated using additional information but not all (answering RQ3.3). To assess the number of inherent differences that are not attributed to tool-specific issues, we tried to eliminate the differences, and we managed to do so in many cases by adjusting or filtering the measurements (see Table 3.8). However, the remaining differences still caused deviations in the coverage values, though they were much lower than the differences for the unmodified tools: at most 0.64% in the total coverage (see Table 3.9). These results show that with a careful tool design, more predictable results could have been achieved, but the full alignment of the different tools seems practically impossible. Since it is not expected from a user to make such corrective actions in the first place, as general advice, users should examine the particular working methods of the tool and be aware of its limitations. Our list of possible reasons for the differences may be used as a guideline on how to avoid and workaround the inaccuracies of the bytecode level instrumentation tools with respect to the source code instrumentation approaches, and in particular to the tools we investigated.

In the last part of our experiments, we checked how the differences in the coverage measurement influenced the results of an application that used coverage as its input (RQ3.4). We applied different test case selection and prioritization algorithms which were all based on the coverage values computed by the two tools. We found that the coverage differences had various influences on the results of the algorithms; the impact was dependent on the different properties of the subject programs and the algorithms themselves (answering RQ3.4). However, for example, the most popular test prioritization algorithm, *additional with resets*, might produce a low correlation of 0.476 between the results of the two tools, which indicates that any practical application or research based on a tool with such inaccuracies imposes a high risk of the validity of the results (see Table 3.11).

We systematically searched for correlations between the subject program properties (modularity, method, and test case numbers), raw measurement values (total and per-test case coverages, coverage differences), and the application results (correlations, reduction rates), but we did not find notable dependencies that could be generalized. It seems that the influence of coverage difference on the applications is subject and algorithm dependent. For example, the average Hamming distance between the individual coverage vectors of MapDB and Lang is very different, 3.062% and 0.014%, respectively. Yet, the correlations between their prioritized lists using the *additional with resets* strategy are similar, 0.815 and 0.751. The programs Time and Netty, which produce very similar average Hamming distances (0.157% and 0.155%) but different correlations of the prioritization (0.672 and 0.476) are examples of the opposite relation. The effect is that the impacts of the inaccuracies in the coverage measurement are unpredictable, hence special care should be taken if code coverage is not used only as a general test completeness measure, but as a base for more complex analysis.

3.6.2 Threats to Validity

The main aim of this work was to investigate the effects of the different instrumentation techniques on code coverage measurement results. We applied empirical measurements using eight subject programs and two specific tools (we started the investigation with four tools). This raises the question of how generalizable the results are to other tools using similar techniques. The subjects were selected from different domains and had different sizes (both in terms of code and tests), but were all actively developed community software.

The two final tools we selected for the detailed examination were among the most widely used coverage tools representing the two instrumentation approaches, and they were mature and actively developed. We carefully analyzed the data in the preliminary experiments from Section 3.4.2 and concluded that there was not a big difference among the candidates from our shortlist in either category. However, limiting the detailed analysis to two tools might impose a threat to the generalizability of the results to other tools. When interpreting the results, we tried to separate the tool-specific issues from the approach-specific ones and the results of source code instrumentation with Clover were verified with manual instrumentation.

A possible threat is that we slightly modified the instrumentation process of JaCoCo by adding a test execution listener that detected the start and the end of the execution of a test case. The results obtained with this modification may not directly translate to the coverage results everyday users would experience with the stock version of JaCoCo. However, we compared the results of the unmodified JaCoCo measurement to our version in terms of actually covered program elements and found no significant differences.

Our experiments showed results with respect to method-level coverage analysis. Generalization to other granularities such as components or statements may not directly be possible.

3.7 Conclusions

The results have shown that even at the method level, significant differences occur between the bytecode and the source code level instrumentation measurements. This confirms the results of some related work (*e.g.,* [53, 4]). Some of the differences can be eliminated, but some cannot, or their elimination would not be practical. These differences, when used in different testing applications, will undoubtedly influence the application results. But the kind and level of influence cannot be generally predicted, as it depends on the subject program and the application itself. A small difference in coverage may be amplified at the application level, and a big coverage difference may have a minor impact.

In conclusion, we may say that the discrepancies between the different instrumentation approaches might but not necessarily influence code coverage applications. It is thus safe to treat source-code-based instrumentation as the correct approach to code coverage measurement, despite its disadvantages (which are summarized in Section 2.1.3). Our results indicate that bytecode instrumentation may have serious disadvantages in terms of the accuracy of the results. The list of possible reasons for the differences may be used as a guideline on how to avoid and work around the inaccuracies of the tools. This can then help assess the level of risk of measurement inaccuracies in particular applications of code coverage measurement.

Thesis I:

1. I worked on the overview of theoretical differences in code coverage measurement tools for Java.
2. I took part in the collection, categorization, testing, and selection of code coverage measurement tools.
3. I also took part in the collection, configuration, and selection of Java programs on which the experiments were executed.
4. I measured and analyzed the differences in code coverage of Java bytecode and source code instrumentation tools.
5. I worked on the systematic investigation of discrepancies in coverage data and their causes, and helped develop fixes and recommendations for the correction of the issues.
6. I analyzed the effects of the found differences on coverage-based applications, namely test selection, and test prioritization.

Response to challenges. Regarding challenge **C1** this thesis provides results that show the quantity of code coverage differences from different aspects. To address the problems related to challenge **C2** the effects and causes of these differences were investigated, and suggestions were made on how some of the differences could be eliminated.

Publications. A preliminary analysis of the differences between bytecode and source code instrumentation was published at the *International Conference on Software Analysis, Evolution and Reengineering* (SANER'16) [c3]. An extension of this work with more deeper and precise analysis of both the causes of differences and the effects of them was published in the *Software Quality Journal* (SQJ'19) [j2].

Applications. The results and experiences of this work helped me in developing a custom instrumentation tool that is used as a basis by several former and ongoing works, for example, the call-chain-based coverage data that is utilized in Thesis III (Chapter 5) was also collected using this tool.

Part II

Fault Localization

4 Interactive Fault Localization

4.1 Introduction

It seems that automatic SBFL methods require external information – not just the program spectra and test case outcomes – to improve on state-of-the-art performance and be more suitable in practical settings. In this work, we propose a form of an *Interactive Fault Localization* approach, called *iFL*. In traditional SBFL, the developer has to investigate several locations before finding the faulty code elements, and all the knowledge they a priori have or acquire during this process is not fed back into the SBFL tool. In our approach, the developer interacts with the fault localization algorithm by giving feedback on the elements of the prioritized list. With this work, we aim at bringing closer the applicability of SBFL methods to practice by involving users' knowledge to the process.

We build on our observations, intuitions and experiences, and we hypothesize that a programmer, when presented with a particular code element, in general has a strong intuition whether any other elements belonging to the same containing higher level code entity should be considered in fault localization. With this intuition, developers can also make a decision ("judge") about the code snippets associated with the item they are currently examining. This allows them to narrow down the search space (*i.e.,* set of the suspicious code elements) more efficiently, which could speed up finding the bug. For example, when users go through the ranked list of suspicious methods, in addition to the examined code element, they could have knowledge about its class, which information can be "fed back" into *iFL* to modify the suspiciousness value of other methods in that class or even exclude items to be examined. This way, larger code parts can be repositioned in their suspiciousness in the hope to reach the faulty element earlier. Other interactive approaches have been proposed by researchers as well [26, 31, 8, 51, 54, 55, 57, 50], but to our knowledge, similar contextual information about higher level entities has not yet been leveraged.

We evaluated the approach in two sets of experiments. First, we used *simulation* to predict the effect of interactivity. We simulated user actions during hypothetical fault finding in well-known bug benchmarks, and measured the Expense metric improvements with respect to the following traditional SBFL formulae: Tarantula [40], Ochiai [2], and DStar [99]. We relied on two benchmarks: artificial defects from the SIR repository [17] and real defects from Defects4J [41]. Results show that the method can significantly improve the fault localization efficiency: in both benchmarks, for 32-57% of the faults their ranking position is reduced from beyond the 10th position to between the 1-10th position. Taking into account all the defects, the localization efficiency in terms of Expense improved on average by 71-79%. For reference, we implemented a closely related interactive FL algorithm proposed by Gong et al. [26], called TALK, in our simulation framework. We compared the performance of *iFL* to TALK on the real faults from Defects4J, and found that *iFL* has a significant advantage over TALK. We also modelled user imperfection, which was rarely studied in related interactive SBFL research. We addressed this aspect from two viewpoints: the user's knowledge and confidence. Experiments simulating these two factors show that *iFL* can outperform a traditional non-interactive SBFL method notably even at low user confidence and knowledge levels.

In the second stage, we performed a quantitative evaluation of the successfulness of *iFL* usage by *real users*. We invited students and professional programmers to solve a set of fault localization tasks using the implementation of the *iFL* approach in a controlled experiment. The goal was to find out whether using the tool shows actual benefits in terms of finding more bugs or finding them more quickly, and this also showed promising results. This experiment also helped us better understand the developers' thought processes and the weaknesses of the approach, and gave us possible directions for future enhancements.

In summary, our contributions are the following.

1. We introduced *iFL*, a novel context-aware interactive fault localization method, embedded in a flexible interactive fault localization framework.

2. We implemented a simulated user and performed experiments on both artificial and real faults. The latter has not yet been studied in interactive fault localization research.

3. We compared the results of our *iFL* experiments to a previously defined algorithm specified in [26].

4. We provide an analysis of two dimensions of user imperfection: knowledge and confidence, which was marginally addressed in previous literature.

5. We implemented *iFL* as an Eclipse plug-in that enables interactive fault localization on Java systems at method level granularity.

6. We performed an empirical study involving real users to compare the fault localization efficiency with and without using the *iFL* approach.

4.2 Motivating Example

For illustration, consider the example in Table 4.1. This is a part of program `replace` from the SIR benchmark repository, which includes manually seeded faults (this benchmark is often used in SBFL research, although being somewhat outdated). Line 116 is a predicate inside function `dodash`, where an artificial fault is seeded: the relation is changed and the `+1` part is deleted (the original version of the code line is shown in a comment). There are three other functions in this program that closely participate in exposing this particular fault, getccl, omatch and locate. The relevant code lines are shown in Table 4.1, while the call-graph in Figure 4.1 shows the high level relationship of the four functions. Function getpat is first called from the main program which indirectly calls getccl and eventually dodash to calculate and return a value. This value is subsequently passed to change and eventually to omatch and locate where the fault will be manifested in form of failing test cases.

Table 4.1: *Example Code and the Fault Localization Process with a Seeded Fault*

Source code		Test cases					Scores and ranks							
Line	**Code**	**557**	**560**	**855**	**857**	**864**	**0. iteration**		**1. iteration**		**2. iteration**		**3. iteration**	
93	`void dodash(delim, src, i, dest, j, maxset)`	•	•	•	•	•	0.658	(23.)	0.658	(20.)	0.658	(7.)	0.658	(5.)
115	`   else if ((isalnum(src[*i - 1])) && (isalnum(src[*i + 1]))`	•	•	•	•		0.677	(14.)	0.677	(12.)	0.677	(5.)	0.677	(4.)
116	`   &&(src[*i - 1] > src[*i])) { //faulty version`	•	•	•	•		0.707	(11.)	0.707	(9.)	0.707	(2.)	0.707	(1.)
116	`   //&&(src[*i - 1] <= src[*i + 1])) { //original version`													
118	`      for (k = src[*i-1]+1; k<=src[*i+1]; k++)`	•	•	•	•		0.707	(12.)	0.707	(10.)	0.707	(3.)	0.707	(2.)
122	`      *i = *i + 1;`	•	•	•	•		0.707	(13.)	0.707	(11.)	0.707	(4.)	0.707	(3.)
123	`   }`													
131	`bool getccl(arg, i, pat, j)`	•	•	•	•	•	0.658	(24.)	0.658	(21.)	0.658	(8.)	0	
144	`   } else`													
145	`      junk = addstr(CCL, pat, j, MAXPAT);`			•	•	•	0.709	(10.)	0.709	(8.)	0.709	(1.)	0	
305	`bool locate(c, pat, offset)`	•	•	•	•	•	0.762	(5.).	0.762	(3.)	0		0	
313	`   flag = false;`	•	•	•	•	•	0.762	(6.)	0.762	(4.)	0		0	
314	`   i = offset + pat[offset];`	•	•	•	•	•	0.762	(7.)	0.762	(5.)	0		0	
315	`   while ((i > offset)) {`	•	•	•	•	•	0.762	(8.)	0.762	(6.)	0		0	
317	`      if (c == pat[i]) {`	•	•	•	•	•	0.765	(4.)	0.765	(2.)	0		0	
318	`         flag = true;`		•		•	•	0.677	(15.)	0.677	(13.)	0		0	
319	`         i = offset;`		•		•	•	0.677	(16.)	0.677	(14.)	0		0	
320	`      } else`													
321	`         i = i - 1;`	•	•	•	•	•	0.768	(3.)	0.768	(1.)	0		0	
322	`   }`													
323	`   return flag;`	•	•	•	•	•	0.762	(9.)	0.762	(7.)	0		0	
327	`bool omatch(lin, i, pat, j)`	•	•	•	•	•								
366	`               if (locate(lin[*i], pat, j + 1))`			•	•	•	0.811	(1.)	0		0		0	
367	`                  advance = 1;`					•	0.665	(18.)	0		0		0	
368	`               break;`			•	•	•	0.811	(2.)	0		0		0	
	Pass/Fail Status	**P**	**F**	**F**	**F**	**P**								

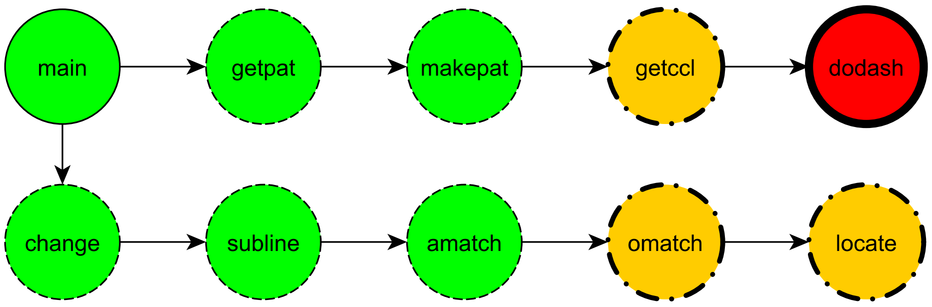


Figure 4.1: *Call-graph of the example program*

Table 4.1 also shows the coverage relationship between some typical test cases and the code elements in question, which expose different behavior with respect to the suspicious elements. We can see that there are passing and failing test cases, and that they are exercising different parts of the program. The faulty statement is traversed both by passing and failing test cases. The fourth column (*0. iteration*) of Table 4.1 corresponds to the suspiciousness scores computed by the Tarantula method[1] along with the ranking position of the elements

[1]We used Tarantula in this example because it is easy to use in the explanation, but other SBFL techniques provide similar relative ranks.

(the ranking position is arbitrary in the case of ties in the scores). There are several lines in functions getccl, omatch and locate that have higher scores than the faulty one from dodash, which will push it farther in the rank, in particular to the 11th-13th place (in the actual implementation, ties are handled so that the average position among the elements with the same value will be used, in this case 12th).

We can explain the failing of SBFL in this case as follows. Recall the Tarantula formula [40] for a code element s:

$$T(s) = \frac{\frac{ef(s)}{ef(s)+nf(s)}}{\frac{ef(s)}{ef(s)+nf(s)} + \frac{ep(s)}{ep(s)+np(s)}},$$

where the functions $ef(s)$, $nf(s)$, $ep(s)$ and $np(s)$ count the number of test cases that execute s and fail, do not execute s and fail, execute s and pass, and do not execute s and pass, respectively. Table 4.2 shows the four basic statistics for lines 116 (the actual fault), 366 (one of the most suspicious statements in the initial ranking) as well as 145 and 321 (the two most suspicious statements in intermediate iterations of our algorithm, which will be presented shortly). We can observe that all failing test cases are exercising statement 116 (30/30), while only (25/30) statement 366. This, in itself, would make the first statement more suspicious, however, the counts for the passing test cases will change the result. In particular, a lot more passing test cases exercise statement 116 (2280/5511) than statement 366 (1066/5511). In other words, there are comparably more *coincidentally correct* tests [61] for the actual faulty statement than for the other, and despite the correct ordering in terms of failing test cases, the final score will flip their relationship.

Table 4.2: *Basic SBFL Statistics for the Example Program*

Line	*ef*	*ep*	*nf*	*np*	Tarantula score
116	30	2 280	0	3 231	0.707
145	25	1 882	5	3 629	0.709
321	30	1 662	0	3 849	0.768
366	25	1 066	5	4 445	0.811

4.3 Interactive Fault Localization

Our approach to improve SBFL is to leverage the background and acquired knowledge of the developers about the system being debugged outside their current focus – the currently investigated code element. We build on our observations, intuitions and experiences, and we hypothesize that a programmer, when presented with a statement from a particular function, in general has an intuition whether any other statements in that function should be considered in fault localization. Or, in a different setting, the programmer is assumed to be able to decide (in certain cases) whether the whole class is faulty or not, if presented with one of its methods. Example situations when such decisions could be made include when the element is known to have been reviewed or otherwise tested recently, it was examined in a previous debugging session, class members follow the same pattern such as getters-setters, etc.

In our approach, we call this information the *contextual* knowledge, which can be fed back to the *iFL* engine. More precisely, we define the **context of an investigated code element** as **the other elements of its enclosing higher level syntactic entity**. For example, in the case of a statement, its context are all the other statements belonging to its function. A context of a function is its enclosing class, and so on.

Suppose that developers are performing SBFL and start with the highest ranked element, statement 366 (see columns "Scores and ranks" in Table 4.1). They look at the function this statement belongs to and conclude that it is not likely to contain the fault (because it was not changed recently, or they examined it in a previous debugging session, etc.). This knowledge is then fed back to the *iFL* engine, which in turn reduces the suspiciousness scores for all contained elements to 0, sending other highly ranked elements to the end of the list. Then, the next most suspicious element – statement 321 of function `locate` – is given to the users. Again, the developers decide based on contextual knowledge that this function is not suspicious, so the engine reduces the scores of all contained statements to 0. This is repeated for line 145 as well in the next iteration. Consequently, several elements are pushed to the end of the list, moving the faulty one, statement 116, to the next rank position. This terminates the fault localization process with success. The effort required to locate the fault was reduced from 12 steps to only 5 (3 steps for removing the three functions and two steps in the final iteration to select the middle one from the three elements with the same suspiciousness score).

Figure 4.2 shows a conceptual overview of our approach. The process starts by calculating an initial rank based on an arbitrary traditional SBFL approach *e.g.,* Tarantula. The elements are then shown to the user starting from the beginning of the list, and the *iFL* engine is waiting for user feedback. The user investigates the recommended element and gives one of the following answers: 1. fault is found, 2. element is not faulty, neither its context, 3. element is not faulty, but the fault is somewhere within the context, or 4. don't know.

Based on the feedback from the user, the *iFL* engine performs the following actions. In the case of (1), the process terminates, while at (4) it is continued as usual with the next suspicious element (this means that in the worst case when the developer has no background knowledge, the method falls back to the pure SBFL approach). In the remaining two cases, the *iFL* engine makes adjustments to the suspiciousness scores, recalculates the ranking and shows the next element from the new list to the user in the next iteration. Note that answer (4) and its corresponding action could mean different things in different settings, and they could be implemented in various ways depending on the actual setting. For example, in an IDE where the suspicious elements are displayed in a list or a table (4) symbolizes that the developer looks at an element and chooses not to interact with it.

There may be different strategies to make the mentioned adjustments, such as applying proportional reductions or increases to the scores, which are different for the context and other parts of the system, etc. Presently, we follow this approach: in the case of (2), the whole context (*i.e.,* function) gets 0 score, while for (3) everything but the context is reduced to 0.

Since there are no increases in the suspiciousness scores, a mistake in the answer made by the developer can move the faulty element towards the end of the ranking which could make the fault finding efficiency even worse than not using the approach. It is safe to assume that developers will not be free of mistakes and completely knowledgeable about the system; in

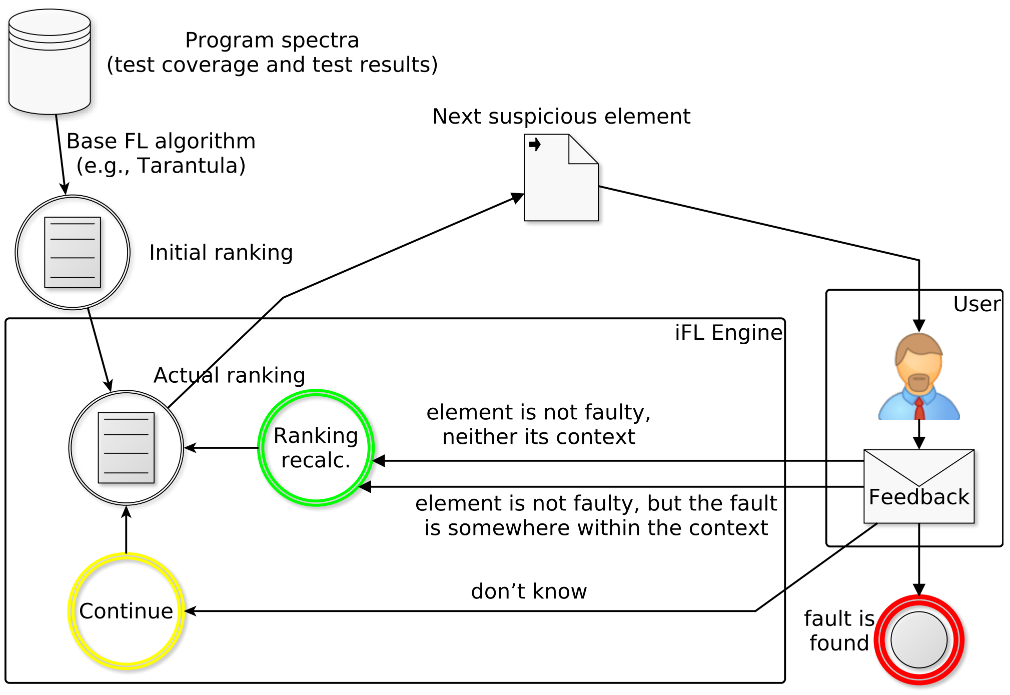


Figure 4.2: *Basic process of Interactive Fault Localization*

some cases, they may not be able to provide additional information on the context of a code element, or they can make wrong decisions. Therefore, part of our research goals in this work is to verify what is the performance of the method when the user does exhibit some imperfection properties (this is our **RQ4.1.3** discussed below).

4.4 Evaluation Goals

We verified the effectiveness of the Interactive Fault Localization approach in two stages. First, we performed an empirical study using *simulated* users (Section 4.5). Next, we compared *iFL* to a closely related interactive approach, called TALK, proposed by Gong et al. [26]. For this, we re-implemented the TALK algorithm in our framework, and we evaluated its performance on real faults with simulated users (Section 4.5.5). These studies were followed by another empirical study involving *real* users (Section 4.6).

The study with simulated users enables large scale and automated experimentation with different faults from existing benchmarks, and predicting the expected effectiveness in real life scenarios. This approach has been followed by most of the related research, *e.g.,* Gong et al. [26] and Hao et al. [31], but we also perform measurements by simulating various degrees of user imperfection, which is a novelty compared to previous studies. On the other hand, evaluation with real users provides direct results about the usefulness of the approach, although only for a limited number of fault finding scenarios.

More precisely, the goal of the first part of the evaluation was the following. *With simulated users, how much improvement in localization effectiveness, in terms of elements to be inspected, can we achieve with iFL over a traditional non-interactive SBFL method and an interactive approach?* We have the following Research Questions for this part of the evaluation:

RQ4.1.1 What improvement can we observe with *iFL* on artificial faults from the SIR repository?

RQ4.1.2 What improvement can we observe with *iFL* on real faults from the Defects4J repository?

RQ4.1.3 How *iFL* compares to another interactive approaches on real faults from the Defects4J repository?

RQ4.1.4 How sensitive *iFL* is to user imperfections?

The goal of the second part of the evaluation was the following. *Given actual fault finding tasks with real users, is it true that users with access to an implementation of iFL in their development environment are able to find more bugs or find them more quickly compared to a control group who did not have access to iFL?* We formulate the following Research Questions for the second part of the evaluation with real users:

RQ4.2.1 Is it true that users could find more bugs with *iFL* than users without access to the method?

RQ4.2.2 Is it true that users could find bugs more quickly with *iFL* than users without access to the method?

RQ4.2.3 How do real users subjectively evaluate the *iFL* method and its implementation in the development environment?

Eventually, the answers to the questions above could help us design new elements into the *iFL* approach and new features for the tool implementation.

4.5 Results with Simulated Users

4.5.1 Experiment Setup

To answer research questions **RQ4.1.1–RQ4.1.4**, we relied on two sets of benchmarks: the SIR repository which contains mostly artificial faults and Defects4J, a benchmark consisting of real faults. These two benchmarks are different also in terms of their size and complexity so we will perform fault localization at different granularity levels. For SBFL, we selected three algorithms: Tarantula [40], Ochiai [2], and DStar [99], which have been reported to be the most successful in different settings [71], and are often referred in literature. With this choice we wanted to verify if the actual algorithm has any impact on the effectiveness of the approach.

Regarding the user responses and *iFL* engine actions, for **RQ4.1.1** and **RQ4.1.2** we follow a relatively simple but strict approach (there are no intermediate, partial or uncertain responses and actions, in other words, we simulate a hypothetical *perfect* user). The perfect user is perfectly sure about her decisions, which means out of the four possible responses explained in Section 4.3, we will not use the fourth one, “don’t know”. The perfect user always recognizes the faulty method. She also has a perfect knowledge about whether the fault is in the context of the inspected element. Furthermore, the mentioned strategy for the actions will be employed, that is, reducing either the whole context or everything but the context to 0. Experiments of user imperfection that answer **RQ4.1.4**, including the "don't know" answer, are presented separately in Section 4.5.6.

We implemented the required components of the *iFL* system according to these settings on different granularities for the two benchmarks, and executed it using all available bugs. The simulated user component works so that it takes the elements from the ranked list starting from the first one, compares their context to the context of the known fault and generates the corresponding answers until the faulty element is reached.

4.5.1.1 Implementation Details of Talk

To answer **RQ4.1.3** We have implemented the TALK algorithm in our simulation framework based on the pseudo codes and descriptions available in [26]. The core of this algorithm is a loop which waits for user feedback. In the loop two rules are utilized to apply adjustments to the scores of code elements. The first rule is triggered when a code element is labelled as clean by the user (the authors call these elements symptoms). This rule aims to find the root cause of such symptoms. When a root cause is found, a portion of the symptom's original score is transferred to the root cause. The second rule is applied when a code element is labelled as faulty. In these cases, the score of the code elements which are covered by the smallest (in terms of the number of covered code elements) test case $t_{\min}$ is increased by a factor $\mathcal{K}_s$. $\mathcal{K}_s$ is set to a value which ensures that in the next few iterations of the algorithm the code elements covered by $t_{\min}$ are displayed at the beginning of the list that is shown to the user.

Note that our evaluation considers only the first faulty code element that is found. Consequently, our simulation stops when the first faulty code element is found, therefore the second rule is never applied in our experiments.

4.5.2 Evaluation Method

To compare the *iFL* method to a traditional SBFL approach, we will use the approach presented in Section 2.2.5, *i.e.,* we will compute Expense metrics for both approaches and compare them in terms of improvement relative to traditional SBFL, and we calculate and present *enabling improvements* as well. Since in each iteration of the approach one block of code is decided upon in one step, we will count each iteration as an equivalent of one rank position for calculating Expense. The amount of improvement will then be calculated for each defect and suitable averages will be produced.

4.5.3 Results for Seeded Faults

To answer **RQ4.1.1**, seven small C/C++ programs from the Software-artifact Infrastructure Repository (SIR) [17] were included in the experiments, which are the so-called "Siemens" suite. This benchmark contains seeded faults, and both the original and faulty versions are available. The subject programs are listed in Table 4.3. Column 2 shows the size of the programs in lines of code (LOC) including the comment and empty lines, along with the number of executable code elements (CE) for which coverage information could be obtained. In column 3, the number of functions in the program is given (this corresponds to the context in *iFL*). The number of test cases in the test suite is presented in column 4, while the 5th one contains the number of available faulty versions (each version has exactly one fault in it).

Note that, the last column of Table 4.3 shows the number of defects we were able to use in the experiments: 1) we filtered out versions where there were multiple faulty code elements; 2) we omitted faults where the coverage tool was unable to record coverage in, *e.g.,* headers and macros; 3) we omitted cases where the suspiciousness score of the faulty code element assigned by the actual SBFL technique was zero. The latter issue was present in only a few cases and slightly differently in the three SBFL methods we investigated, which resulted in different number of cases we used for subject printtokens2 (also see Table 4.4). However, since there were no ways to improve fault localization efficiency in these cases, this did not impact the measurement results. For preparing the raw data for the *iFL* experiments including the code coverage information and test case results, the tools [24] and [87] were used.

Table 4.3: *Details of Subject Programs from SIR*

Program	**LOC (CE)**	**Functions**	**Tests**	**Faults**	**Suitable faults**
printtokens	726 (277)	18	4 130	7	1
printtokens2	570 (262)	19	4 115	10	7
replace	564 (400)	21	5 542	32	22
schedule	412 (225)	18	2 650	9	2
schedule2	374 (198)	16	2 710	10	4
tcas	173 (95)	9	1 608	41	31
totinfo	565 (187)	7	1 052	23	18
Total	**3 384 (1 644)**	**108**	**21 807**	**132**	**85**

Table 4.4 shows the improvements *iFL* was able to achieve on SIR. The performance of the original SBFL algorithms can be seen in column 4, which we used as the reference to evaluate *iFL*. Both absolute and relative versions of the Expense measure are provided. For each of the SBFL techniques, a summarization line is provided with the corresponding average values. The three techniques achieved similar results, Ochiai being slightly better than the other two. On average, it prioritized the faulty code elements roughly to the 20th place, which means that on average 13% of the executable code elements must be examined to find the faulty one. Ochiai was followed by DStar (19.90 – 13.24%) and then by Tarantula (24.85 – 15.43%).

Column 5 contains the same data for *iFL*. The average Expense measures are notably better than for the original algorithm. Ochiai performs slightly better in this case as well with an Expense of 5.78 (3.75%) on average. This means, that in this case a programmer would need only about six steps (5.78) to find the fault on average. DStar was the second best in this comparison with 5.83 (3.80%), but Tarantula was similar as well 6.86 (4.25%). In terms of relative improvement (Column 6), Tarantula achieved the best results: 17.99 steps (11.19%). It was followed by DStar with 14.11 (9.47%) and Ochiai was last with 14.01 (9.34%). Column 7 of the table contains a summary of improvements in terms of relative changes in the Expense values, expressed in percentage (that is, the difference over the SBFL base value). For all techniques, the improvement is notable, **71-72%**.

The last four columns of Table 4.4 summarize the *enabling improvements iFL* achieved on the SIR benchmark. Here, the number of faults (and their relative ratio) are presented

Table 4.4: *iFL Improvements on SIR*

Alg.	Program	Faults	$E(E')$			Impr.	Enabling improvements			
			Score	iFL	Diff.		$[\infty, 10) \to [10, 5)$	$[\infty, 10) \to [5, 1]$	$[10, 5) \to [5, 1]$	Total
DStar	printtokens	1	5.00 (1.81%)	2.00 (0.72%)	-3.00 (-1.08%)	60.00%	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)
	printtokens2	5	23.00 (8.78%)	6.30 (2.40%)	-16.70 (-6.37%)	72.61%	1 (20.00%)	1 (20.00%)	0 (0.00%)	2 (40.00%)
	replace	22	13.05 (3.26%)	3.66 (0.91%)	-9.39 (-2.35%)	71.95%	1 (4.55%)	6 (27.27%)	5 (22.73%)	12 (54.55%)
	schedule	2	6.25 (2.78%)	2.50 (1.11%)	-3.75 (-1.67%)	60.00%	0 (0.00%)	0 (0.00%)	1 (50.00%)	1 (50.00%)
	schedule2	4	66.62 (33.59%)	13.25 (6.68%)	-53.38 (-26.92%)	80.11%	0 (0.00%)	1 (25.00%)	0 (0.00%)	1 (25.00%)
	tcas	31	20.26 (21.32%)	5.39 (5.67%)	-14.87 (-15.65%)	73.41%	11 (35.48%)	11 (35.48%)	1 (3.23%)	23 (74.19%)
	totinfo	18	18.78 (10.04%)	8.06 (4.31%)	-10.92 (-5.84%)	58.14%	7 (38.89%)	1 (5.56%)	0 (0.00%)	8 (44.44%)
		83	**19.90 (13.24%)**	**5.83 (3.80%)**	**-14.11 (-9.47%)**	**70.91%**	**20 (24.10%)**	**20 (24.10%)**	**7 (8.43%)**	**47 (56.63%)**
Ochiai	printtokens	1	5.00 (1.81%)	2.00 (0.72%)	-3.00 (-1.08%)	60.00%	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)
	printtokens2	7	19.14 (7.31%)	4.79 (1.83%)	-14.36 (-5.48%)	75.00%	1 (14.29%)	1 (14.29%)	0 (0.00%)	2 (28.57%)
	replace	22	13.09 (3.27%)	3.66 (0.91%)	-9.43 (-2.36%)	72.05%	1 (4.55%)	6 (27.27%)	5 (22.73%)	12 (54.55%)
	schedule	2	6.25 (2.78%)	2.50 (1.11%)	-3.75 (-1.67%)	60.00%	0 (0.00%)	0 (0.00%)	1 (50.00%)	1 (50.00%)
	schedule2	4	66.62 (33.59%)	13.25 (6.68%)	-53.38 (-26.92%)	80.11%	0 (0.00%)	1 (25.00%)	0 (0.00%)	1 (25.00%)
	tcas	31	20.32 (21.39%)	5.42 (5.70%)	-14.90 (-15.69%)	73.33%	11 (35.48%)	11 (35.48%)	1 (3.23%)	23 (74.19%)
	totinfo	18	19.00 (10.16%)	8.28 (4.43%)	-10.92 (-5.84%)	57.46%	8 (44.44%)	0 (0.00%)	0 (0.00%)	8 (44.44%)
		85	**19.74 (13.07%)**	**5.78 (3.75%)**	**-14.01 (-9.34%)**	**70.95%**	**21 (24.71%)**	**19 (22.35%)**	**7 (8.24%)**	**47 (55.29%)**
Tarantula	printtokens	1	5.00 (1.81%)	2.00 (0.72%)	-3.00 (-1.08%)	60.00%	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)
	printtokens2	7	30.71 (11.72%)	7.21 (2.75%)	-23.50 (-8.97%)	76.51%	1 (14.29%)	0 (0.00%)	0 (0.00%)	1 (14.29%)
	replace	22	19.18 (4.80%)	4.70 (1.18%)	-14.48 (-3.62%)	75.47%	2 (9.09%)	6 (27.27%)	5 (22.73%)	13 (59.09%)
	schedule	2	10.75 (4.78%)	5.50 (2.44%)	-5.25 (-2.33%)	48.84%	1 (50.00%)	0 (0.00%)	0 (0.00%)	1 (50.00%)
	schedule2	4	77.38 (39.02%)	14.25 (7.18%)	-63.12 (-31.84%)	81.58%	0 (0.00%)	1 (25.00%)	0 (0.00%)	1 (25.00%)
	tcas	31	21.65 (22.78%)	5.58 (5.87%)	-16.06 (-16.91%)	74.22%	9 (29.03%)	11 (35.48%)	1 (3.23%)	21 (67.74%)
	totinfo	18	26.00 (13.90%)	10.33 (5.53%)	-15.69 (-8.39%)	60.36%	4 (22.22%)	0 (0.00%)	1 (5.56%)	5 (27.78%)
		85	**24.85 (15.43%)**	**6.86 (4.25%)**	**-17.99 (-11.19%)**	**72.42%**	**17 (20.00%)**	**18 (21.18%)**	**7 (8.24%)**	**42 (49.41%)**

falling in the three categories of enabling improvements. According to the last column, the total ratio of improvements that turned out to be enabling is quite large, around **49-57%**. More importantly, most of these improvements are those that bring the faulty code element from outside of top 10 into top 10 (Column 8) or, even better, into top 5 (Column 9). The two programs on which *iFL* produces the highest rate of enabling improvements are tcas and replace. Interestingly, tcas is the smallest and replace is the largest program in our set, which may indicate that there is no connection between improvement rate and program size.

Answer to RQ4.1.1: In the case of SIR programs containing seeded faults, the Ochiai method produced the best results from SBFL techniques. Compared to it, *iFL* achieved **71% improvement** in Expense, and resulted in **47 enabling improvements**, which corresponds to **55% of the faults**.

4.5.4 Results for Real Faults

In the field of Interactive Fault Localization, there is an emerging need for studies that go beyond the size and complexity of the SIR repository. A recent study by Pearson et al. [71] investigates existing techniques both on SIR and Defects4J, showing that the latter repository has quite different properties when it comes to the performance of various SBFL techniques. Here, we present our measurement results with *iFL* on defects from the Defect4J repository [41], to answer **RQ4.1.2**.

Defects4J is a database and extensible framework which provides a high-level interface to real defects, and has been widely used in software testing research [6, 114, 42, 85]. The version (v2.0.0) of the dataset that we used contains 17 open source Java programs and 835 bugs in total. For *iFL* experiments, we extended the Defects4J framework with our Java agent-based code-coverage measurement tool[2]. This tool attaches to the JVM and utilizes on-

[2]The extended framework is available at: `https://github.com/Frenkymd/defects4j/tree/chain`

the-fly bytecode instrumentation to collect different levels of coverage information. Defects4J provides the fix for each bug as a patch set. We created a static analyzer tool which uses JavaParser [37] to analyze the patch sets and provides information about the changed code elements. Then, using the patch sets and the information provided by the static analyzer we were able to create change sets that contain data about which methods were affected by which bug fixes. There are some bugs in this benchmark where the change set becomes empty. The reason for this is that the patch set contains only additions of code elements *i.e.,* the changed elements did not exist in the faulty version of the program. Obviously, non existent methods are not considered by any FL approach, therefore we excluded these cases from our experiments. In addition, there are some bugs where the suspiciousness score of the faulty code elements assigned by the actual SBFL technique was zero or undefined – we excluded these cases too. Note that, different algorithms could produce zero or undefined scores in different settings, therefore the number of excluded bugs could change from algorithm to algorithm. In our experiment we encountered 37 cases where only failing tests exercised the faulty code elements. In these cases, the denominator of the DStar formula $(ep + (ef + nf) - ef)$ evaluates to zero $(0 + (n + 0) - n = 0)$ and due to the division by zero the score is undefined. The main properties of programs from the Defects4J dataset can be seen in Table 4.5, and the number of bugs we could use in the experiments is shown in Table 4.6 (Faults column).

Table 4.5: *Main Properties of Programs Used from Defects4J*

Program	Bugs	Size (LOC)	Tests	Methods	Classes
Chart	26	96 382	2 193	5 227	472
Cli	39	1 936	94	149	19
Closure	174	90 694	7 911	8 392	1151
Codec	18	2 584	206	234	19
Collections	4	26 409	15 393	3 532	422
Compress	47	6 740	73	437	53
Csv	16	0 806	54	83	11
Gson	18	5 378	720	620	106
JacksonCore	26	15 871	206	795	73
JacksonDatabind	112	42 964	1 098	3 546	447
JacksonXml	6	4 679	138	293	35
Jsoup	93	2 546	139	327	41
JxPath	22	19 372	308	1 287	131
Lang	64	21 778	2 291	2 353	158
Math	106	84 317	4 378	6 350	818
Mockito	38	10 517	1 378	1 454	291
Time	26	27 795	4 041	3 612	204
Total	**835**	**460 768**	**40 621**	**38 691**	**4 451**

In this experiment, the granularity of fault localization was elevated to the method level because of two reasons. First, we could use statement level granularity as well, but the benchmark contains larger and real programs, and even on method level it includes a large number of code elements. Second, we wanted to check how does the algorithm behave on this level and if there is a significant difference in terms of effectiveness to the other benchmark. Our feedback-based algorithm needed adjustment as well: the basic elements are changed from source code lines to methods, and the context is changed from functions to classes. Otherwise, the main steps of the *iFL* process (from Figure 4.2) including the responses and actions were the same as with the statement-level granularity. The measurements themselves followed the same steps as we used for SIR in Section 4.5.3, and the results will be presented in the same way in this section as well. Therefore, detailed explanation of the structure of tables will be omitted.

Table 4.6: *iFL Improvements on Defects4J*

Alg,	Program	Faults	***E*(*E*′)**			Impr.	Enabling improvements			
			Avg rank	**Avg rank w *iFL***	**Diff.**		$[\infty, 10) \rightarrow [10, 5)$	$[\infty, 10) \rightarrow [5, 1]$	$[10, 5) \rightarrow [5, 1]$	**Total**
DStar	Chart	22	9.14 (0.20%)	2.91 (0.06%)	-6.23 (-0.14%)	68.16%	1 (4.55%)	3 (13.64%)	4 (18.18%)	8 (36.36%)
	Cli	37	14.45 (5.85%)	4.62 (1.97%)	-9.82 (-3.88%)	68.01%	6 (16.22%)	5 (13.51%)	6 (16.22%)	17 (45.95%)
	Closure	173	89.45 (1.18%)	15.89 (0.21%)	-73.64 (-0.96%)	82.32%	34 (19.65%)	17 (9.83%)	14 (8.09%)	65 (37.57%)
	Codec	16	6.25 (1.61%)	3.25 (0.81%)	-3.00 (-0.80%)	48.00%	0 (0.00%)	1 (6.25%)	1 (6.25%)	2 (12.50%)
	Compress	46	15.73 (1.61%)	4.71 (0.51%)	-11.02 (-1.10%)	70.08%	3 (6.52%)	6 (13.04%)	5 (10.87%)	14 (30.43%)
	Csv	16	5.97 (5.03%)	3.06 (2.26%)	-3.25 (-2.99%)	54.45%	0 (0.00%)	1 (6.25%)	5 (31.25%)	6 (37.50%)
	Gson	16	18.72 (2.53%)	8.75 (1.19%)	-10.09 (-1.35%)	53.92%	3 (18.75%)	0 (0.00%)	0 (0.00%)	3 (18.75%)
	JacksonCore	22	9.11 (0.94%)	3.73 (0.36%)	-5.39 (-0.58%)	59.10%	2 (9.09%)	2 (9.09%)	4 (18.18%)	8 (36.36%)
	JacksonDatabind	96	60.59 (1.37%)	10.67 (0.24%)	-49.94 (-1.13%)	82.41%	19 (19.79%)	8 (8.33%)	9 (9.38%)	36 (37.50%)
	JacksonXml	4	23.00 (7.78%)	6.00 (2.03%)	-17.00 (-5.75%)	73.91%	1 (25.00%)	0 (0.00%)	1 (25.00%)	2 (50.00%)
	Jsoup	88	31.53 (3.25%)	8.20 (0.81%)	-23.36 (-2.44%)	74.07%	12 (13.64%)	10 (11.36%)	8 (9.09%)	30 (34.09%)
	JxPath	21	53.48 (4.08%)	12.71 (0.97%)	-40.76 (-3.11%)	76.22%	3 (14.29%)	2 (9.52%)	2 (9.52%)	7 (33.33%)
	Lang	52	4.90 (0.24%)	2.40 (0.12%)	-2.54 (-0.12%)	51.76%	7 (13.46%)	1 (1.92%)	0 (0.00%)	8 (15.38%)
	Math	95	10.49 (0.27%)	3.62 (0.10%)	-6.88 (-0.17%)	65.58%	12 (12.63%)	11 (11.58%)	9 (9.47%)	32 (33.68%)
	Mockito	35	24.24 (2.24%)	6.84 (0.60%)	-17.40 (-1.64%)	71.77%	2 (5.71%)	5 (14.29%)	0 (0.00%)	7 (20.00%)
	Time	25	19.06 (0.55%)	4.88 (0.14%)	-14.46 (-0.42%)	75.87%	2 (8.00%)	1 (4.00%)	3 (12.00%)	6 (24.00%)
		764	**39.28 (1.74%)**	**8.36 (0.49%)**	**-30.96 (-1.26%)**	**78.82%**	**107 (14.01%)**	**73 (9.55%)**	**71 (9.29%)**	**251 (32.85%)**
Ochiai	Chart	25	7.96 (0.18%)	2.64 (0.06%)	-5.32 (-0.12%)	66.83%	1 (4.00%)	3 (12.00%)	4 (16.00%)	8 (32.00%)
	Cli	39	14.09 (5.74%)	4.68 (2.01%)	-9.41 (-3.73%)	66.79%	6 (15.38%)	5 (12.82%)	7 (17.95%)	18 (46.15%)
	Closure	173	90.25 (1.19%)	15.92 (0.21%)	-74.33 (-0.97%)	82.36%	36 (20.81%)	16 (9.25%)	13 (7.51%)	65 (37.57%)
	Codec	16	6.34 (1.63%)	3.28 (0.82%)	-3.06 (-0.81%)	48.28%	0 (0.00%)	1 (6.25%)	1 (6.25%)	2 (12.50%)
	Collections	1	1.00 (0.03%)	1.00 (0.03%)	0.00 (0.00%)	-0.00%	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)
	Compress	47	15.72 (1.61%)	4.72 (0.52%)	-11.00 (-1.09%)	69.96%	2 (4.26%)	6 (12.77%)	5 (10.64%)	13 (27.66%)
	Csv	16	5.97 (5.03%)	3.06 (2.26%)	-3.25 (-2.99%)	54.45%	0 (0.00%)	1 (6.25%)	5 (31.25%)	6 (37.50%)
	Gson	16	18.91 (2.55%)	8.94 (1.22%)	-10.09 (-1.35%)	53.39%	3 (18.75%)	0 (0.00%)	0 (0.00%)	3 (18.75%)
	JacksonCore	25	8.48 (0.87%)	3.80 (0.36%)	-4.68 (-0.51%)	55.19%	2 (8.00%)	1 (4.00%)	5 (20.00%)	8 (32.00%)
	JacksonDatabind	101	58.49 (1.32%)	10.62 (0.24%)	-47.89 (-1.08%)	81.88%	18 (17.82%)	8 (7.92%)	9 (8.91%)	35 (34.65%)
	JacksonXml	5	18.60 (6.29%)	5.00 (1.69%)	-13.60 (-4.60%)	73.12%	1 (20.00%)	0 (0.00%)	1 (20.00%)	2 (40.00%)
	Jsoup	89	31.26 (3.22%)	8.03 (0.79%)	-23.25 (-2.43%)	74.37%	14 (15.73%)	9 (10.11%)	8 (8.99%)	31 (34.83%)
	JxPath	22	51.66 (3.94%)	11.05 (0.84%)	-40.61 (-3.10%)	78.62%	4 (18.18%)	3 (13.64%)	2 (9.09%)	9 (40.91%)
	Lang	61	4.51 (0.22%)	2.23 (0.11%)	-2.31 (-0.11%)	51.27%	6 (9.84%)	1 (1.64%)	1 (1.64%)	8 (13.11%)
	Math	104	10.00 (0.26%)	3.51 (0.10%)	-6.50 (-0.16%)	65.02%	13 (12.50%)	11 (10.58%)	10 (9.62%)	34 (32.69%)
	Mockito	35	24.47 (2.25%)	6.96 (0.61%)	-17.51 (-1.64%)	71.57%	3 (8.57%)	5 (14.29%)	0 (0.00%)	8 (22.86%)
	Time	26	18.40 (0.53%)	4.81 (0.14%)	-13.87 (-0.40%)	75.34%	2 (7.69%)	0 (0.00%)	3 (11.54%)	5 (19.23%)
		801	**37.93 (1.68%)**	**8.10 (0.48%)**	**-29.86 (-1.21%)**	**78.71%**	**111 (13.86%)**	**70 (8.74%)**	**74 (9.24%)**	**255 (31.84%)**
Tarantula	Chart	25	13.00 (0.30%)	3.16 (0.07%)	-9.84 (-0.23%)	75.69%	0 (0.00%)	3 (12.00%)	3 (12.00%)	6 (24.00%)
	Cli	39	15.06 (6.09%)	4.76 (2.05%)	-10.31 (-4.04%)	68.43%	5 (12.82%)	5 (12.82%)	7 (17.95%)	17 (43.59%)
	Closure	173	98.36 (1.30%)	16.35 (0.22%)	-82.01 (-1.08%)	83.38%	32 (18.50%)	17 (9.83%)	14 (8.09%)	63 (36.42%)
	Codec	16	6.59 (1.68%)	3.44 (0.85%)	-3.16 (-0.83%)	47.87%	0 (0.00%)	1 (6.25%)	1 (6.25%)	2 (12.50%)
	Collections	1	1.00 (0.03%)	1.00 (0.03%)	0.00 (0.00%)	-0.00%	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)
	Compress	47	17.36 (1.81%)	5.04 (0.55%)	-12.32 (-1.26%)	70.96%	2 (4.26%)	7 (14.89%)	5 (10.64%)	14 (29.79%)
	Csv	16	5.97 (5.03%)	3.06 (2.26%)	-3.25 (-2.99%)	54.45%	0 (0.00%)	1 (6.25%)	5 (31.25%)	6 (37.50%)
	Gson	16	18.88 (2.55%)	9.06 (1.23%)	-10.06 (-1.35%)	53.31%	3 (18.75%)	0 (0.00%)	0 (0.00%)	3 (18.75%)
	JacksonCore	25	8.34 (0.82%)	3.66 (0.33%)	-4.68 (-0.50%)	56.12%	2 (8.00%)	2 (8.00%)	3 (12.00%)	7 (28.00%)
	JacksonDatabind	101	58.97 (1.33%)	10.63 (0.24%)	-48.36 (-1.09%)	82.00%	18 (17.82%)	10 (9.90%)	8 (7.92%)	36 (35.64%)
	JacksonXml	5	18.60 (6.29%)	5.00 (1.69%)	-13.60 (-4.60%)	73.12%	1 (20.00%)	0 (0.00%)	1 (20.00%)	2 (40.00%)
	Jsoup	89	31.98 (3.28%)	8.22 (0.81%)	-23.76 (-2.47%)	74.30%	13 (14.61%)	10 (11.24%)	8 (8.99%)	31 (34.83%)
	JxPath	22	42.27 (3.22%)	10.59 (0.81%)	-31.68 (-2.41%)	74.95%	3 (13.64%)	3 (13.64%)	4 (18.18%)	10 (45.45%)
	Lang	61	5.20 (0.25%)	2.46 (0.12%)	-2.77 (-0.13%)	53.31%	7 (11.48%)	0 (0.00%)	0 (0.00%)	7 (11.48%)
	Math	104	9.89 (0.25%)	3.57 (0.10%)	-6.33 (-0.15%)	64.03%	14 (13.46%)	10 (9.62%)	11 (10.58%)	35 (33.65%)
	Mockito	35	27.40 (2.52%)	8.07 (0.69%)	-19.33 (-1.83%)	70.54%	4 (11.43%)	5 (14.29%)	0 (0.00%)	9 (25.71%)
	Time	26	19.71 (0.57%)	5.17 (0.15%)	-14.73 (-0.43%)	74.73%	1 (3.85%)	0 (0.00%)	4 (15.38%)	5 (19.23%)
		801	**40.07 (1.74%)**	**8.33 (0.49%)**	**-31.77 (-1.26%)**	**79.27%**	**105 (13.11%)**	**74 (9.24%)**	**74 (9.24%)**	**253 (31.59%)**

With *iFL* we achieved high improvements compared to all the three fault localization metrics. The first part of Table 4.6 shows average ranking improvements (Diff. column) and its ratio compared to the number of all possible code elements (*i.e.,* Java methods).

The Expense measures for the original SBFL methods are quite different than for the SIR programs, they range from 1-98 steps, which is much more than the average on small programs, however, the relative measures are smaller, 1.68-1.74% on average. This is due to the significantly larger number of program elements in this benchmark (despite the higher granularity level). *iFL* achieved a notable improvement with this benchmark as well, as can be seen from columns 5 and 6 of the table. The difference is between 30 and 32 positions on average, but given the large total number of elements, the change in percentages is modest. In this case, Ochiai produced the best initial ranking and the best final Expense measures with *iFL* as well. The relative improvement (column 7) is higher than for the SIR benchmark, it is about **79%** for each SBFL technique. Practically, this means that on average the *iFL* approach could potentially save 79% of the human effort.

More importantly, in the case of large programs and real defects there are many cases when *iFL* achieved *enabling improvements*. Detailed data is shown in the second part of Table 4.6. Overall, *iFL* had **251-255 (32-33%)** enabling improvements, which is slightly worse than for the SIR programs. In most cases, *iFL* brings the faulty elements into the top-10 or top-5 range from outside of top-10. These are the cases where the original SBFL produced very bad Expense results initially. Compared to SIR, the lower number of test cases may be one reason for this phenomenon, but finding the actual causes needs more investigation. Note that, this benchmark contains much larger programs and that the original Expense measures were typically much higher as well.

Answer to RQ4.1.2: In the case of Defects4J experiments with real faults, the Ochiai method produced the best results from traditional techniques. Relative to it, *iFL* achieved **79% improvement** in Expense, and produced **255 enabling improvements**, corresponding to **32% of the faults in this benchmark**.

4.5.5 Results for Real Faults with the Talk Algorithm

In this section, we present the results of the replication study in which we re-implemented the TALK algorithm proposed by [26].

Note that we could not replicate the original experiment to its full extent. We did our best to acquire the data and the most detailed description of the experiments from the original paper. However, we could only work with what was published. In order to accommodate the highest number of bugs we used the latest version of Defects4J. Since the approach for coverage data collection was not mentioned in the original paper, we could not reuse the exact same measurement methods. We also used method level granularity, which could have affected the results. In addition, we ran into some issues while executing TALK on the two largest subject programs from Defects4J. Unfortunately, because of the complexity of the algorithm we were not able to execute TALK on Closure and on some bugs of JacksonDatabind.

We followed the same approach to evaluate TALK that we presented at the beginning of Section 4.5, and we used the same benchmark on which we evaluated *iFL* in Section 4.5.4. However, as knowledge and confidence cannot be interpreted for the TALK algorithms, we did not check the effect of these factors in the replication experiments.

As published, the baseline performance of TALK in terms of improvement is between -14% and 60% (12.29% on average) with Ochiai, -25% and 46% (11.71%) with Jaccard and -35% and 72% (13.37%) with Tarantula. Table 4.7 shows the performance of the re-implemented

Talk algorithm on the Defects4J benchmark in the same fashion as Table 4.6 in Section 4.5.4. As it can be seen, we obtained varying results. The best performance is achieved using DStar. In this setting, improvements range from about -6% up to 10% and 1.56% overall. In the case of Ochiai and Tarantula, Talk is behind its traditional SBFL counterparts by a slight margin, -0.58% and -7.20% respectively. The maximal improvement (16%) is achieved on Chart with Tarantula. However, in the case of some projects Talk performs significantly worse than Ochiai and Tarantula. Also, the number of enabling improvements is very limited compared to *iFL*.

Table 4.7: *Talk Improvements on Defects4J*

Alg,	Program	Faults	$E(E')$			Impr.	Enabling improvements			
			Avg rank	Avg rank w Talk	Diff.		$[\infty, 10) \rightarrow [10, 5)$	$[\infty, 10) \rightarrow [5, 1]$	$[10, 5) \rightarrow [5, 1]$	Total
DStar	Chart	22	9.14 (0.20%)	8.59 (0.19%)	-0.55 (-0.02%)	5.97%	0 (0.00%)	0 (0.00%)	1 (4.55%)	1 (4.55%)
	Cli	37	14.45 (5.85%)	13.97 (5.65%)	-0.47 (-0.20%)	3.27%	1 (2.70%)	1 (2.70%)	1 (2.70%)	3 (8.11%)
	Codec	16	6.25 (1.61%)	6.25 (1.60%)	0.00 (-0.00%)	-0.00%	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)
	Compress	46	15.73 (1.61%)	16.05 (1.72%)	0.33 (0.11%)	-2.07%	1 (2.17%)	0 (0.00%)	1 (2.17%)	2 (4.35%)
	Csv	16	5.97 (5.03%)	5.75 (4.97%)	-0.22 (-0.06%)	3.66%	0 (0.00%)	0 (0.00%)	1 (6.25%)	1 (6.25%)
	Gson	16	18.72 (2.53%)	18.38 (2.48%)	-0.34 (-0.05%)	1.84%	1 (6.25%)	0 (0.00%)	0 (0.00%)	1 (6.25%)
	JacksonCore	22	9.11 (0.94%)	9.68 (0.98%)	0.57 (0.04%)	-6.23%	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)
	JacksonDatabind	41	32.76 (0.80%)	29.44 (0.72%)	-3.32 (-0.07%)	10.13%	0 (0.00%)	1 (2.44%)	1 (2.44%)	2 (4.88%)
	JacksonXml	4	23.00 (7.78%)	22.50 (7.61%)	-0.50 (-0.17%)	2.17%	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)
	Jsoup	88	31.53 (3.25%)	30.46 (3.09%)	-1.07 (-0.16%)	3.41%	0 (0.00%)	1 (1.14%)	5 (5.68%)	6 (6.82%)
	JxPath	21	53.48 (4.08%)	56.29 (4.29%)	2.81 (0.21%)	-5.25%	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)
	Lang	52	4.90 (0.24%)	5.02 (0.25%)	0.12 (0.01%)	-2.35%	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)
	Math	95	10.49 (0.27%)	10.69 (0.27%)	0.21 (0.00%)	-1.96%	2 (2.11%)	0 (0.00%)	1 (1.05%)	3 (3.16%)
	Mockito	35	24.24 (2.24%)	24.10 (2.22%)	-0.14 (-0.02%)	0.59%	1 (2.86%)	0 (0.00%)	0 (0.00%)	1 (2.86%)
	Time	25	19.06 (0.55%)	19.32 (0.56%)	0.26 (0.01%)	-1.36%	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)
		536	**18.78 (1.92%)**	**18.48 (1.89%)**	**-0.29 (-0.03%)**	**1.56%**	**6 (1.12%)**	**3 (0.56%)**	**11 (2.05%)**	**20 (3.73%)**
Ochiai	Chart	25	7.96 (0.18%)	6.76 (0.15%)	-1.20 (-0.03%)	15.08%	0 (0.00%)	0 (0.00%)	1 (4.00%)	1 (4.00%)
	Cli	39	14.09 (5.74%)	14.60 (6.10%)	0.51 (0.36%)	-3.64%	0 (0.00%)	2 (5.13%)	1 (2.56%)	3 (7.69%)
	Codec	16	6.34 (1.63%)	6.22 (1.55%)	-0.12 (-0.08%)	1.97%	1 (6.25%)	0 (0.00%)	0 (0.00%)	1 (6.25%)
	Collections	1	1.00 (0.03%)	1.00 (0.03%)	0.00 (0.00%)	-0.00%	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)
	Compress	47	15.72 (1.61%)	16.23 (1.74%)	0.51 (0.13%)	-3.25%	0 (0.00%)	1 (2.13%)	1 (2.13%)	2 (4.26%)
	Csv	16	5.97 (5.03%)	6.44 (5.58%)	0.47 (0.54%)	-7.85%	0 (0.00%)	0 (0.00%)	1 (6.25%)	1 (6.25%)
	Gson	16	18.91 (2.55%)	19.00 (2.56%)	0.09 (0.01%)	-0.50%	1 (6.25%)	0 (0.00%)	0 (0.00%)	1 (6.25%)
	JacksonCore	25	8.48 (0.87%)	9.56 (0.96%)	1.08 (0.09%)	-12.74%	0 (0.00%)	0 (0.00%)	2 (8.00%)	2 (8.00%)
	JacksonDatabind	44	31.52 (0.77%)	29.48 (0.72%)	-2.05 (-0.04%)	6.49%	0 (0.00%)	1 (2.27%)	1 (2.27%)	2 (4.55%)
	JacksonXml	5	18.60 (6.29%)	18.20 (6.16%)	-0.40 (-0.13%)	2.15%	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)
	Jsoup	89	31.26 (3.22%)	30.11 (3.09%)	-1.15 (-0.12%)	3.67%	0 (0.00%)	2 (2.25%)	3 (3.37%)	5 (5.62%)
	JxPath	22	51.66 (3.94%)	56.14 (4.28%)	4.48 (0.34%)	-8.67%	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)
	Lang	61	4.51 (0.22%)	4.79 (0.23%)	0.28 (0.02%)	-6.18%	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)
	Math	104	10.00 (0.26%)	10.72 (0.28%)	0.72 (0.02%)	-7.16%	0 (0.00%)	0 (0.00%)	1 (0.96%)	1 (0.96%)
	Mockito	35	24.47 (2.25%)	24.61 (2.26%)	0.14 (0.01%)	-0.58%	1 (2.86%)	0 (0.00%)	0 (0.00%)	1 (2.86%)
	Time	26	18.40 (0.53%)	18.81 (0.54%)	0.40 (0.01%)	-2.19%	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)
		571	**17.95 (1.83%)**	**18.05 (1.88%)**	**0.10 (0.05%)**	**-0.58%**	**3 (0.53%)**	**6 (1.05%)**	**11 (1.93%)**	**20 (3.50%)**
Tarantula	Chart	25	13.00 (0.30%)	10.92 (0.24%)	-2.08 (-0.06%)	16.00%	0 (0.00%)	0 (0.00%)	1 (4.00%)	1 (4.00%)
	Cli	39	15.06 (6.09%)	15.96 (6.66%)	0.90 (0.58%)	-5.96%	1 (2.56%)	1 (2.56%)	0 (0.00%)	2 (5.13%)
	Codec	16	6.59 (1.68%)	6.38 (1.65%)	-0.22 (-0.04%)	3.32%	1 (6.25%)	0 (0.00%)	0 (0.00%)	1 (6.25%)
	Collections	1	1.00 (0.03%)	1.00 (0.03%)	0.00 (0.00%)	-0.00%	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)
	Compress	47	17.36 (1.81%)	18.02 (1.94%)	0.66 (0.13%)	-3.80%	0 (0.00%)	1 (2.13%)	1 (2.13%)	2 (4.26%)
	Csv	16	5.97 (5.03%)	6.44 (5.43%)	0.47 (0.40%)	-7.85%	0 (0.00%)	0 (0.00%)	1 (6.25%)	1 (6.25%)
	Gson	16	18.88 (2.55%)	20.56 (2.77%)	1.69 (0.22%)	-8.94%	1 (6.25%)	0 (0.00%)	0 (0.00%)	1 (6.25%)
	JacksonCore	25	8.34 (0.82%)	12.02 (1.15%)	3.68 (0.33%)	-44.12%	0 (0.00%)	0 (0.00%)	2 (8.00%)	2 (8.00%)
	JacksonDatabind	42	28.26 (0.70%)	29.99 (0.74%)	1.73 (0.05%)	-6.11%	0 (0.00%)	1 (2.38%)	1 (2.38%)	2 (4.76%)
	JacksonXml	5	18.60 (6.29%)	16.20 (5.48%)	-2.40 (-0.82%)	12.90%	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)
	Jsoup	89	31.98 (3.28%)	31.85 (3.28%)	-0.13 (-0.00%)	0.40%	1 (1.12%)	4 (4.49%)	2 (2.25%)	7 (7.87%)
	JxPath	22	42.27 (3.22%)	50.41 (3.84%)	8.14 (0.62%)	-19.25%	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)
	Lang	61	5.20 (0.25%)	5.82 (0.29%)	0.62 (0.03%)	-11.99%	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)
	Math	104	9.89 (0.25%)	12.45 (0.32%)	2.56 (0.07%)	-25.91%	0 (0.00%)	0 (0.00%)	1 (0.96%)	1 (0.96%)
	Mockito	35	27.40 (2.52%)	28.59 (2.61%)	1.19 (0.09%)	-4.33%	1 (2.86%)	0 (0.00%)	0 (0.00%)	1 (2.86%)
	Time	26	19.71 (0.57%)	20.94 (0.61%)	1.23 (0.04%)	-6.24%	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)
		569	**18.13 (1.88%)**	**19.43 (2.00%)**	**1.31 (0.12%)**	**-7.20%**	**5 (0.88%)**	**7 (1.23%)**	**9 (1.58%)**	**21 (3.69%)**

Answer to RQ4.1.3: Regarding Expense, TALK achieves -0.29 (**1.56%**), 0.10 (**-0.58%**) and 1.31 (**-7.20%**) improvement on average compared to DStar, Ochiai and Tarantula respectively, which is worse than *iFL*'s **78.72%**, **78.71%** and **79.27%**. Considering enabling improvements, TALK produces **20-21** such improvements which correspond to the **3.50-3.73%** of the faults, which is significantly less than the 251-255 (**31.59-32.85%**) that *iFL* produces.

4.5.6 Effect of User Imperfections

To answer **RQ4.1.4**, in this section we investigate to what extent user imperfection affects the results of our method. This phenomenon is only marginally addressed in interactive fault localization literature. Hao et al. [31] tackled the problem of user imperfection by incorporating two factors into their approach and analysis. They introduce a parameter which approximates the confidence of developers by acting as a scaling factor on suspiciousness modifications. Also, they define the concept of accuracy rate to represent the probability that the developer makes correct estimations. Li et al. [54, 55] used a similar approach to simulate the reliability of the users by modifying the automated oracle in their experiment such that it gives erroneous answers on a configurable rate. Both studies consider a limited range (5-30%, 50%) of the various factors that impact the effectiveness, and they conclude that various factors modelling user imperfections indeed had impact on the effectiveness results, but not that significant which would invalidate their findings. We provide a similar, but more detailed analysis of user imperfection by experimenting with two factors that may influence the validity of simulated users:

Confidence Level This factor indicates how much confidence we have in the user for providing reliable answers. We model confidence by applying a proportional decrease of the scores instead of nullation as with the base algorithm. The *iFL* engine was modified to scale down the suspiciousness of appropriate code elements proportionally to the confidence level: the new score s' is calculated from the original s using confidence level c as: $s' = (1 - c)s$. Here, $c = 0$ means no confidence in which case the original scores remain, and with perfect confidence, $c = 1$, nullation will be performed.

Knowledge Level In our model, the knowledge of the users means the rate at which they can make informed decisions about the context. Thus, knowledge is modelled by the user's ability to give meaningful answers about the context as a whole. This factor was implemented by letting the user choose the "don't know" response randomly with a frequency that is inversely proportional to the knowledge level. This means that a perfect knowledge, $k = 1$, allows no "don't know" responses, while with no knowledge at all, $k = 0$, every answer will be of this type, falling back to the base FL algorithm.

These two factors were designed and implemented solely for the experiments for answering **RQ4.1.4**. If both factors are set to 1, we will obtain the base approach we used for research questions **RQ4.1.1-RQ4.1.3**. Any combination of values are interesting to observe to what extent they are influencing the effectiveness of the *iFL* method. We re-executed our experiments on both datasets with confidence and knowledge levels set between 20% and 100% in 10% steps. We decided to ignore values below 20% because they simulate an unlikely situation in which the user is very incompetent. Due to the random factor that was introduced by the implementation of the knowledge level, we repeated each measurement 100 times and used the average data that was collected during the iterations.

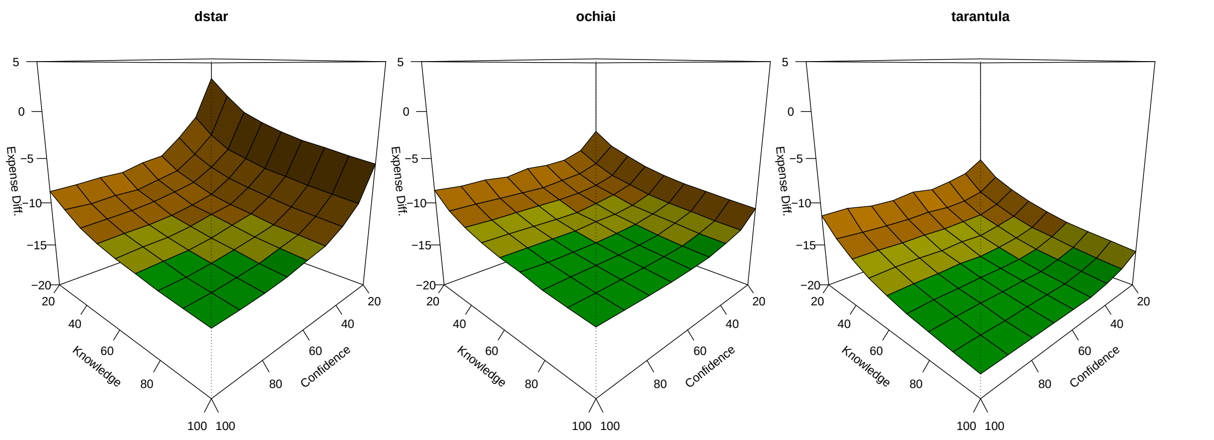


Figure 4.3: *iFL improvement with different knowledge and confidence levels on SIR*

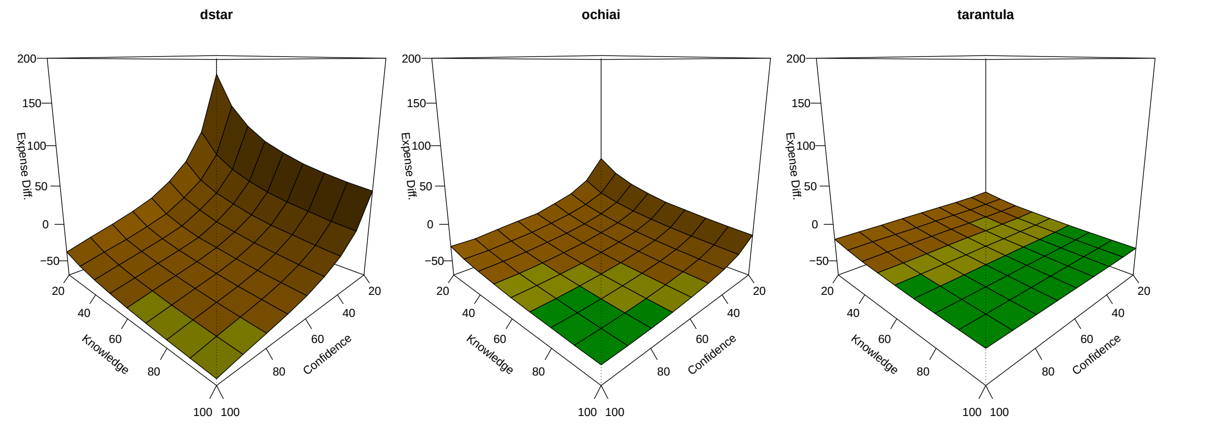


Figure 4.4: *iFL improvement with different knowledge and confidence levels on Defects4J*

Improvement levels, in terms of absolute Expense difference, can be seen in Figures 4.3 and 4.4. Each point on the 3D surface represents a different configuration of knowledge and confidence. The near and far edges of the 3D cube mean the perfect and almost completely incompetent users, respectively. Results show that both knowledge and confidence affect the performance of *iFL*, though to various extent. The algorithm is sensitive to both factors when DStar is used as the base suspiciousness score, however it is relatively stable with Ochiai and Tarantula. We can observe that confidence and knowledge have different effects. While the algorithm scales almost linearly with knowledge, lack of confidence causes performance loss on a near exponential rate. This seems to be aligned with the everyday observation that high confidence combined with low knowledge is a worse situation than a low confidence with high knowledge scenario.

Table 4.8 shows the approximate lowest knowledge and confidence levels allowable to limit the performance loss of *iFL* to at most 10 or 20% (reduction to 80 or 90% of the gain of the base *iFL* algorithm). These requirements were calculated by setting one of the factors to 100%, *e.g.,* the last column in the last row means that confidence can be as low as 20% (while knowledge is set to 100%), but *iFL* still provides 80% of its benefits compared to the best case scenario on Defects4J using Tarantula (except DStar on Defects4J).

Table 4.8: *Requirements for Keeping 80-90% of Improvement*

		90%		80%	
		Know.	Conf.	Know.	Conf.
SIR	DStar	60%	60%	40%	40%
	Ochiai	60%	40%	40%	30%
	Tarantula	60%	30%	40%	20%
D4J	DStar	70%	90%	50%	80%
	Ochiai	60%	70%	50%	50%
	Tarantula	60%	20%	40%	20%

User imperfection is more realistic when both factors are changed at the same time. For the minimum gain levels 80-90%, consider the areas of the surfaces in Figures 4.3 and 4.4 marked with different shades of yellow and green. Points satisfying the 80 or 90% gain level are positioned in the lower part of the surface. To keep the desired gain level, a tradeoff between confidence and knowledge may be made but when both aspects of user imperfection exceed the roughly 30-40% levels, the overall gain will be above the 80% of the gain of the perfect user.

Answer to RQ4.1.4: User imperfection, as modelled by our experiments, affects the results of the context aware *iFL* algorithm differently. *iFL* is sensitive to the knowledge and the confidence factors as well. Our experiments show that even very low confidence (**20-30%**) and knowledge levels (**30-40%**) suffice to keep **80%** of the improvements.

4.6 Quantitative Results with Real Users

4.6.1 Experiment Setup

To answer research questions **RQ4.2.1-RQ4.2.3**, we performed a user study involving real programmers and asking them to solve real fault localization tasks within an IDE. For the experiment setup, we reused parts of the methodology followed by Parnin and Orso [70] and Le et al. [49].

Participants 36 software engineering students were invited for participation on a voluntary basis, of which 22 were BSc (undergraduate) and 14 were MSc (graduate) students. Only the top-performing students were invited from the class of about 230 based on their previous scholastic performance in the relevant subjects. Only students with sufficient knowledge of Java, Eclipse, and its debugging features were included. The programming experience of the participants was between 0.5–6 years, on average 2.5 years. 23 participants had at least 1 year programming experience working in industry. Three groups ($G1 - G3$) have been formed randomly, each having approximately the same ratio of BSc and MSc students.

To diversify the experiment we also invited professional programmers from the software development teams of our university. We excluded everyone who had some relation to the topic of this paper and from the 4 volunteers we created group $G4$. The average programming experience in this group was 12.75 years (between 5–18 years).

Tool Support We implemented a prototype tool [t1, p1] as an Eclipse plug-in that implements the basic functionality of *iFL*. Currently, it supports Java projects and fault localization on method granularity. It provides in a window a ranked list of methods with the associated suspiciousness scores, the context (enclosing class), and other information. The user can interact with this list, provide the feedback, make filtering on the scores, navigate to the source, etc.

Task Assignment We designed altogether 8 different fault localization tasks ($A - H$), keeping in mind that participants should be able to solve each task in about 30 minutes including understanding the problem and documenting the solution (it was treated successful if the participant can briefly explain the required fix but no actual implementation was needed). We also wanted to ensure some diversity, so we selected A and E to be small, the rest large programs, B, E, F be simple bugs and the rest more complex, some to have low Tarantula ranks (A, C, D), and the rest high ranks. This information was not told to the participants.

We selected 6 concrete bugs from the Defects4J database (3 from Commons Math and 3 from Joda-Time) and 2 bugs from a student project. One important consideration in the selection was to have bugs where the rank of the faulty code element is small (math-53) and slightly larger (ship-3), however, other aspects were also considered. Such characteristics were the following: **(i)** the number of faulty methods, *i.e.,* there should be a bug where one method is faulty (math-5) and also one where there are multiple faulty methods (time-9) **(ii)** the complexity of the bugfix, *i.e.,* the fix should not be too complicated/complex (since students participated in the experiment) and **(iii)** the similarity of ranks, *i.e.,* the FL algorithms should produce the same result in terms of the ranks of the faulty methods. These properties ensured that we control the most variables during the experiment. In particular, the eight bugs with their ID-s, names and Tarantula ranks were the following: [A] ship-3 (rank 11), [B] math-5 (rank 2), [C] time-9 (rank 7), [D] time-8 (rank 7), [E] ship-1 (rank 2), [F] math-53 (rank 1), [G] time-4 (rank 5), [H] math-4 (rank 2).

Groups $G1 - G3$ have each been assigned 4 different tasks from the 8 available, half of which had to be executed using the *iFL* functionality and the rest without it (feedback was disabled). In order to increase the diversity, we assigned the tasks and tool modes in various combinations to the groups. We reserved two more complex tasks for group $G4$, and tool modes were randomly selected when the participants started their tasks. Table 4.9 shows the task assignments (I: *iFL* used, N: *iFL* not used).

Table 4.9: *Task Assignment*

Group / Task	***A***	***B***	***C***	***D***	***E***	***F***	***G***	***H***
$G1$	N	I	-	N	I	-	-	-
$G2$	-	N	I	-	N	I	-	-
$G3$	I	-	N	I	-	N	-	-
$G4$	-	-	-	-	-	-	I/N	I/N

Experiment Execution The experiment was executed in two sessions with three half hour blocks in each session and a break between the sessions. The first block was dedicated to introduce participants to the experiment, explain the goals, the basic functionality of the tool and other instructions. Then, the four tasks have been performed in the next blocks, where 25 minutes were available to actually perform the task and 5 minutes were reserved for documenting the results and switching to the next task. After each task and for each participant, we recorded the following information: number of minutes for completion if it was successful, the solution, how much was the tool used (*none, little, fully*), how much did it help (*none, little, a lot*), if the tool was not used what method was used instead to find the bug. During the final 30 minutes, the participants were asked to fill a questionnaire about their general impressions and comments: how useful was the approach (on a scale 1-5), actual benefits and drawbacks encountered, further information that could be used as the context and other ideas to improve the tool.

4.6.2 Results for Bug Finding Efficiency

Results regarding the number of completed tasks are presented in Table 4.10.[3] For each task, we include the total number of participants who performed it, using *iFL* support and without it (columns 2, 4 and 6, respectively).[4] Columns *Compl.* show the number of participants who successfully completed the task belonging to the group using the tool and not using it, respectively (also, percentages are given wrt. the number of participants in the corresponding groups).

Table 4.10: *Number of All and Completed Tasks*

Bug	**Overall**		**with *iFL***		**no *iFL***	
	All	**Compl.**	**All**	**Compl.**	**All**	**Compl.**
A - ship-3	24	3 (12%)	12	2 (17%)	12	1 (8%)
B - math-5	24	19 (79%)	12	9 (75%)	12	10 (83%)
C - joda-9	24	14 (58%)	12	5 (42%)	12	9 (75%)
D - joda-8	24	18 (75%)	12	8 (67%)	12	10 (83%)
E - ship-1	25	15 (60%)	13	8 (62%)	12	7 (58%)
F - math-53	23	23 (100%)	12	12 (100%)	11	11 (100%)
Total	144	92 (64%)	73	44 (60%)	71	48 (68%)

The overall success was 64% but it was highly varying across the different tasks. We could not observe any dependence on the program size (A vs. D), but more simpler bugs could be localized by more participants (B vs. E), overall. Initially, it was not our intent, but task A turned out to be the most difficult to solve by the participants. Besides the relative complexity of the bug, this might also be influenced by the fact that it was the first assignment for the participants, who perhaps did not have enough understanding of the tool at that time.

[3]In group $G4$ success rate was 100% for both tasks, and the completion time varied between 10-29 minutes independently from tool usage, but due to the very low number of participants we excluded this group from the evaluation of effectiveness and efficiency.

[4]For E and F, number of group members with and without *iFL* was not equally 12+12 because two workstations had to be exchanged due to technical reasons, which resulted in a slight change to the planned task assignment.

We could not observe any difference in the success rate of participants who used *iFL* compared to those who did not. In particular, the number of participants who successfully solved the tasks is approximately the same, there is even a slight increase in the overall number of cases without tool support. A very slight improvement within the group using the tool can be observed for tasks *A* (small but difficult bug), *E* (complex bug) and *F* (simple bug and high Tarantula rank).

Answer to RQ4.2.1: *iFL* does not seem to help in localizing more bugs. A slight increase in success rate can be observed in the case of complex bugs and when the Tarantula rank is very high.

Table 4.11: *Task Completion Time in Hours, Minutes and Seconds*

Bug	**with *iFL***	**no *iFL***	**Diff.**
A - ship-3	0:16:00	0:25:00	-0:09:00 (-36%)
B - math-5	0:10:33	0:05:48	0:04:45 (82%)
C - joda-9	0:08:00	0:14:53	-0:06:53 (-46%)
D - joda-8	0:15:15	0:21:06	-0:05:51 (-28%)
E - ship-1	0:16:07	0:19:17	-0:03:09 (-16%)
F - math-53	0:12:40	0:09:16	0:03:23 (37%)
Total	**9:30:00**	**11:05:00**	**-1:35:00 (-14%)**

Table 4.11 shows the results we collected about the time required to localize the fault, which was needed for **RQ4.2.2**.

For each bug, we present the average times required for completion over all group members who managed to complete the task. The last column shows the difference of the times (absolute and relative) with respect to the cases without tool support. We could observe a noticeable overall improvement, of **14%**. But, results also show that there is a big variance of the difference across the different tasks: in the case of the Commons Math bugs, *B* and *F*, the tool even resulted in longer completion times, but in the other cases there was 16-46% improvement on average. Both Commons Math bugs were quite easy to understand and to locate the faulty element in them, so it might be the case that the use of *iFL* resulted in such a big overhead that not using the tool was actually more simpler. The overall improvement excluding these two tasks was about **36%** (5:23:00 vs. 8:25:00 total times). We could not observe any relationship between the program size or Tarantula rank and the completion times.

Answer to RQ4.2.2: Using *iFL* reduced the time required to localize the fault, overall by **14%**, except for the cases when the bug was very simple and easy to identify (without these, the improvement was **36%**).

4.6.3 Subjective Evaluation by the Participants

For answering **RQ4.2.3**, participants were asked to fill out a questionnaire that consisted of two parts: short questions about each bug and questions about the approach and tool in general. Table 4.12 includes the data for the first part, the responses per bug (this relates only to tasks where *iFL* was enabled). In more than two-thirds of all cases, participants expressed their opinion regarding the usage and usefulness of *iFL*.

Table 4.12: *Subjective Evaluation of iFL per Bug*

Bug	**Usage**				**Usefulness**			
	no answ.	**none**	**little**	**fully**	**no answ.**	**none**	**little**	**a lot**
A - ship-3	6	0	3	3	6	2	3	1
B - math-5	2	2	6	2	3	1	6	2
C - joda-9	5	2	5	0	5	2	3	2
D - joda-8	5	2	2	3	6	2	1	3
E - ship-1	3	1	5	4	3	2	5	3
F - math-53	1	0	8	3	1	0	7	4
Total	**22 (30%)**	**7 (10%)**	**29 (40%)**	**15 (20%)**	**24 (33%)**	**9 (12%)**	**25 (34%)**	**15 (21%)**

Users responded that they did not use the tool in 7 cases (10%); they used it a little in 29 cases (40%); and relied fully on *iFL* in 15 cases (20%). Overall, participants used the interactive approach at least a little in **44 cases (60%)**. Considering the usefulness of *iFL*, the results were similar. Participants did not find the approach helpful at all in 9 (12%) cases, but in the remaining 25 (34%) and 15 (21%) cases they found that *iFL* aided fault localization at least a little or a lot, respectively (this is **82%** of the cases when participants responded). Experts also agreed on the usefulness of the tool, but they argued that the complexity of the bugs may have an impact on it. However, we did not identify any pattern in the distribution of opinions wrt. differences in bug types.

In the next part of the survey, participants were asked to rate the usefulness of the interactivity for SBFL in general on a scale 1–5 (*1-not useful*, *5-extremely useful*). The results are presented in Fig. 4.5. **Two thirds of the participants (24)** said that *iFL* was useful at least moderately and only 8.3% (3) answered that they did not find it helpful at all.

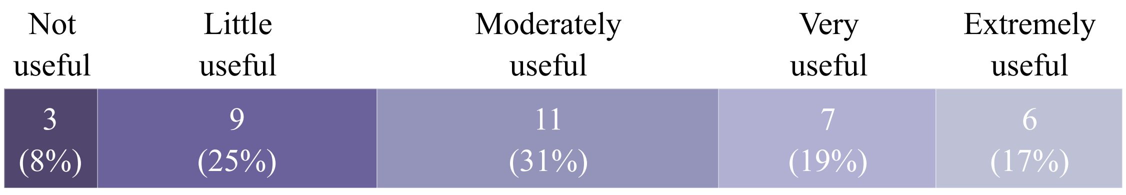


Figure 4.5: *Overall usefulness of the iFL approach*

Participants could also write a textual evaluation of the approach and the tool, in which they could list the advantages and disadvantages they experienced. Some typical benefits mentioned:

> “The tool gives good hints and it can confirm if my idea is good or not”,

> “It is straightforward to navigate between suspicious functions”.

Some disadvantages of the tool mentioned by participants:

"The tool can mislead and so gives an unnecessary overhead",

"We can exclude the actually faulty functions with feedback, which cannot be undone".

In addition, participants could articulate suggestions for using or further developing the tool. Several commented that it would be helpful if the selected method could be automatically opened in a separate window or a view. Also, undo-redo and the dynamic score-update were among the most frequently mentioned missing features. Some commented that for a large set of methods, the traditional search function made it easier to find the appropriate methods.

Professional developers said that the tool provides good starting points for debugging and it also helps focusing their efforts on the most suspicious parts of the source code. However, they also added that more information would be beneficial to make full use of the potential of interactivity and to make decisions about contexts easier. They mentioned the visualization of factors that contribute to the suspiciousness scores (*e.g.,* related tests, especially the failing ones) and provide more detailed information about the bug (*e.g.,* stack traces, call-chains, etc.) in an organized, easily understandable way as the most advantageous improvement.

Answer to RQ4.2.3: For most tasks where they responded, participants found interactive feedback useful to find the bug (**82%**), and **two thirds** of the participants said that in general it was useful at least at a moderate level. Textual responses about the benefits and drawbacks will help us in further developing the approach and tool.

4.7 Discussion

This section contains our general views on the interpretation of the empirical data from previous sections.

4.7.1 Conceptual Evaluation Based on Simulations

As discussed, *iFL* achieved notable improvements on both benchmarks; Table 4.13 summarizes the most important results. For reference, the Talk algorithm proposed by Gong et al. [26] achieves about 10-16% improvement at best on some projects from Defects4J, and it produces 20-21 enabling improvements. DStar and Ochiai behaves very similarly and they produce slightly different results compared to Tarantula. However, DStar is very sensitive to user imperfections. Results are comparable for the two benchmarks, despite the significant differences in their properties including size and the number of tests compared to the number of program elements.

We confirmed the effectiveness of *iFL* with paired t-test and signed Wilcoxon test to check if the differences in the Expense measures between *iFL* and the basic SBFL over the entire dataset are statistically significant. These tests showed that *iFL* results are significantly better than SBFL at 99% confidence level for each SBFL technique and benchmark.

4.7.2 Possible Improvement Based on Experiments with Real Users

The previously mentioned results concerned the validity and usefulness of the *iFL* methodology from a conceptual point of view. However, there are at least two main practical issues which could hinder its application in real life situations. Namely, do developers have enough

Table 4.13: *Main Results of the iFL Algorithm*

Study	Best FL method	Fault type	All faults	Improvement	Enabling improvement
SIR	Ochiai	seeded	85	71%	47 (55%)
Defects4J	Ochiai	real	801	79%	255 (32%)

information about the context (*i.e.,* the surrounding code base) to be able to give adequate feedback for the *iFL* algorithm? Furthermore, developers have to integrate the proposed methodology and its implementation into their daily workflows. The latter could affect several other factors, not only the raw efficiency as we used during the simulation.

Based on the findings in Sections 4.6, we were able to articulate several concrete improvements to the *iFL* methodology and current implementation in the Eclipse IDE to address the above issues. We categorize these into three groups: improvements to the methodology, functional enhancements to the implementation and design and other technical improvements to the tool.

In addition to these issues, several participants explicitly expressed their opinion that without accurate background knowledge of the underlying method (such as score calculation) they would be less trustful in the outcomes of the tool. These kinds of issues are related to multiple aspects of the *iFL* methodology and its implementation. Technical details should be explained in the documentation and calculation details need to be accessible from the tool on demand. We could solve these issues in several different ways. For example, we could use various help features in the IDE, or clarify the concept and principle of score calculation in a detailed training session.

4.7.2.1 Methodology

The main improvements regarding the *iFL* methodology include lowering the granularity level and experimenting with different kinds of contextual data. Currently, in the case of large programs the *iFL* engine works on method level. However, we have seen that users often demand more detailed information during debugging. Since *iFL* is designed to cope with different granularity levels, changing the granularity to line/statement level is fairly straightforward after we eliminate the technical limitations implied by the current coverage measurement approach. Users usually incorporate multiple information sources into their debugging process. Along this idea *iFL* itself could be extended to utilize additional data, *e.g.,* function call contexts, as contextual information to improve the effectiveness.

4.7.2.2 Functional Enhancements

We could aid the integration of *iFL* into the developers' workflow by implementing features which enhance the user experience of the methodology. For example, a possible improvement is automatic score calculation after test execution. So, by running the tests during or after the (bug)repair, we can modify (upgrade) the score values and get a real-time picture of the change in suspiciousness values. These changes can provide useful information for locating the bug or identifying other faulty (*i.e.,* , not yet inspected and corrected) methods.

The degree of trust in the tools or the users' own decisions could impact the willingness of the tools' usage. There may be cases when users approve an interaction and then they regret it afterwards. This could be due to, for example, a "false-click" or because the users have changed their previous judgment or opinion and therefore they want to revoke it. This would be supported by the undo and redo functions, which allow the developers to rollback any *iFL* score changes.

Currently, we do not support other IDEs besides Eclipse, so another important goal is to create a plug-in for other environments such as IntelliJ.

4.7.2.3 Tool Design and Technicalities

Experience has shown that the participants' main source of information is the code itself; hence it would be more useful to "bring the information closer" to the source code. One possible way to do this is to visualize it in the code-editor window, for example by code highlight. This would greatly help in understanding the *iFL*'s results and increase usability. This feature may also help us to reduce the overcrowding of method rank lists and the information in them.

4.8 Threats to Validity

4.8.1 Simulation

The main threat to the validity of the results related to **RQ4.1** is the set of projects and bugs that we used during our experiments. We selected the SIR and Defects4J benchmarks because they are popular in the scene of fault localization research, hence we maximize the comparability of our results to the related works. However, some researchers argue that the projects and bugs in these benchmarks are not representative of real life projects and bugs. We restricted the experiments in this paper to single-fault cases, however this is a current technical restriction, the proposed method itself is not limited in this regard. In addition, multi-fault cases would not have an effect on the results considering the Expense metrics, since (as usual in these experiments) only the faulty code element with the highest rank is considered during the calculation of the evaluation metrics. Considering the experiments related to user imperfections, multi-fault cases may yield different results, because the algorithm may have more ways to modify the scores of faulty and non-faulty code elements as well.

The next threat is that the feedback model of our method is relatively simple, which might not be suitable in a real life scenario. For example, the user might give some more complex feedback, like combining code elements from other contexts or pointing to code elements not suggested in the rank list. Also, as new knowledge is obtained, changes to earlier decisions could be possible. We plan to address this topic in future work.

4.8.2 Quantitative User Study

The first threat that we identified which could impact the validity of the results of our user studies in **RQ4.2** is the time to perform the tasks. We set the time limit to 25 minutes for each task, which may have an effect on the participants' performance, especially if they have less experience. Also, a longer period of time would have affected the performance as well, since the developers could loose concentration as they get tired.

It is also a possibility that the rank of the faulty code element may have influenced our results. Although, differently ranked faults may yield different results, investigating the correlation between the position of faulty code elements and the success rate or time needed to fix the bugs was not an explicit goal of our experiments. In addition, we tried to select bugs with as different ranks as possible while keeping in mind the distribution of the complexity of the tasks. Also, during the design of the tasks we investigated various projects and we selected bugs for which the developers do not need specific domain knowledge. In other words, we simulated the scenario in which the developer has to debug an unknown program. However, some of the participants may had knowledge about the projects or bugs, which could have influenced their performance.

An other threat is the so called Hawthorne effect [20], *i.e.,* the participants might have changed their behavior in response to the awareness of being observed. They may try to figure out the goals of the experiment in which they participate. To mitigate this threat we minimized the amount of information that might have helped the participants determine the goal of the experiment. However, there is still a chance that they were able to deduce the purpose of the experiment.

A further threat is related to the participants of the studies. They were undergraduate and graduate students, and professional developers. We tried to diversify the list of participants by inviting people with different levels of experience, yet they may not be a representative sample of the population of developers. In addition, we tried to design the experimental groups and assign the tasks to them in such a way that the distribution of participants through the groups is balanced.

4.9 Related Work

The closest related works from the other Interactive Fault Localization methods [26, 31, 8, 51, 54, 55] to our approach are the ones that change the ranking of program elements based on the user feedback iteratively [26, 31, 51]. Other than utilizing user feedback, these papers also incorporate the Siemens suite from SIR into their set of subject programs. However, the setup of experiments and metrics used for the evaluation are different in every case. Differences include the set of defects, the total number of code elements, different interpretation of the localization effectiveness metrics, etc. This makes it difficult to compare our results directly to the reported ones in these works, however, we managed to re-implement the work of Gong et al. [26].

For reference, Lei et al. [51] utilize test data generation techniques to automatically produce feedback for interacting with fault localization techniques. They used a very similar metric to ours (E') to measure the relative effectiveness improvement and concluded that the improvement is around 21% on average compared to the 71-72% range achieved by *iFL*. Hao et al. [31] propose a trace-based method which is reported to achieve about 8% in a similar measure; they also showed that about 90% accuracy from the user is needed to improve the base SBFL algorithm.

In the work of Gong et al. [26], the user simply decides whether the statements are faulty or not. Ranking is updated to find the root cause of the fault using the program spectra. Their approach yields about 12-13% absolute improvement in Expense on average over Tarantula and Ochiai on small programs from the SIR repository [2]. For a more precise comparison which includes real bugs, we reimplemented the TALK algorithm proposed by Gong et al. [26] in our simulation framework and re-executed our experiments. We found that TALK improves the rank of the faulty elements by 1-3 positions which translates to 2-6% improvement on Defects4J.

Li et al. [54, 55] uses a concept of contextual knowledge that is similar to ours. They build on the assumption that the semantics of a method wrt. inputs and output is well known by developers. They generate queries and use the feedback to guide the SBFL based recommendation process in a debugging scenario. Also, they considered the correlation between the success rate of their approach and the percentage of erroneous answers to these queries. Bandyopadhyay and Ghosh [8] proposed a method to iteratively predict and remove coincidentally correct test cases based on user feedback .

Fry and Weimer [23] used software- and defect-related features to study human accuracy at locating faults. They found that certain types of bugs are much harder for humans to locate accurately. Also, they identified source code features that can foretell human FL accuracy and proposed formal models of debugging accuracy based on these features.

4.10 Conclusions

In this thesis, we presented *iFL* – a new form of an Interactive Fault Localization approach. This approach extends traditional Spectrum-Based Fault Localization by providing the ability for the developer to interact with the fault localization algorithm. Interaction means giving feedback on the elements of the prioritized list, based on which the suspiciousness scores are adjusted. We exploit the knowledge of the user about the next item in the ranked list (*e.g.,* a statement or a function) and its context (the containing function or class), with which larger amounts of code elements can be repositioned in their suspiciousness.

With the two-staged approach to empirical evaluation of the proposed approach, we tried to investigate the potential in the *iFL* techniques from many different angles, but this might still not represent real life scenarios fully. However, we obtained promising results in both stages. Results with simulated users showed quite big improvements with respect to non-interactive SBFL even in the case when the users exhibit low levels of reliability. In addition to non-interactive approaches, *iFL* also outperformed TALK, a closely related interactive FL algorithm by a huge margin. Although *iFL* does not seem to help in localizing more bugs when it comes to real users, the fault localization times were reduced significantly (except for the very simple cases), and subjective feedback was also mostly positive about our experimental implementation. However, further research is required to find out how the methodology involving the interactivity should be made more intuitive, and the supporting tools more directly related to the source code, the centerpiece of the developer's attention during debugging. This would enable better acceptance by practitioners without extensive training and explanation through use cases.

Thesis II:

1. I worked on the development of the theoretical background of applying the concept of interactive feedback in fault localization.
2. I also worked on the development of the theoretical background of applying the concept of user imperfection factors (confidence and knowledge) in the evaluation of fault localization.
3. I designed and implemented the simulation framework as a basis for testing interactive fault localization approaches.
4. I implemented the *iFL* approach in the simulation framework.
5. I performed experiments using seeded and real faults from the SIR and Defects4J benchmark to evaluate *iFL*.
6. I measured and analyzed the effectiveness and efficiency of *iFL* in the aforementioned simulated environment.
7. I took part in the reimplementation of the Talk algorithm, and the execution of the subsequent comparative experiments and analyses.
8. I also took part in the design and development of the iFL4Eclipse plug-in which implements *iFL* in the Eclipse IDE.
9. I worked on the design, execution and analysis of the user studies.

Response to challenges. Regarding challenge **C3** this thesis shows that by incorporating the additional knowledge of the developers, the effectiveness and the efficiency of debugging process can be significantly improved. In addition to showing the benefits of incorporating additional information, in response to challenge **C4**, this thesis proposes an interactive fault-localization approach which could be tailored to the developers' needs.

Publications. The general ideas from which *iFL* was composed were published at the *International Conference on Software Maintenance and Evolution* (ICSME'20) [c2]. This paper includes the evaluation of the proposed approach in simulated and real world scenarios, but to a limited extent. Later, we extended the evaluation to significantly more faults, as well as, we implemented the Talk algorithm proposed by Gong et al. [26] to evaluate and compare it directly to our approach. In addition, we conducted a more elaborated user-study with a few experienced developers. In this experiment, we utilized the think-aloud method to gain the most accurate insight into the users' thought processes during debugging tasks. The findings of this study were published recently in the *Empirical Software Engineering* (EMSE'22) [j1].

Applications. The *iFL* approach has been implemented in a tool called iFL4Eclipse, that is an Eclipse plug-in which provides developers with all the benefits of the interactive debugging process. The details of this tool can be found in [p1, t1] and on the following websites: [35, 78].

5 Call-Chain-Based Fault Localization

5.1 Introduction

Based on the vast amount of research performed in the field of SBFL, it seems that variations to these basic approaches may yield only marginal improvements, and that perhaps more radical changes in how we approach the problem are required to achieve more significant gains. For example, by combining conceptually different approaches [117], or by involving additional information to the process. Early attempts to incorporate control or data flow information, for instance [77, 32], have not been further developed because it soon became apparent that they are difficult to scale to large programs and real defects.

Beszédes et al. [c1] propose the concept of enhancing traditional SBFL with *function call chains* on which the FL is performed. Function call chains are snapshots of the call stack occurring during execution and as such can provide valuable context to the fault being traced. Call chains (and call stack traces) are artifacts that are well-known by the programmers who perform debugging and can show, for instance, that a function may fail if called from one place and perform successfully when called from another.

In this thesis, we propose a novel SBFL algorithm, that computes ranking on all occurring call chains during execution, and then selects the suspicious functions from these ranked chains using a function-level (*i.e.,* method-level for object-oriented languages like Java) spectrum-based algorithm, Ochiai in particular [2].

Our approach works at a higher granularity than statement-level approaches (previous work suggests that function-level is a suitable granularity for the users [6, 117]). At the same time, we provide more context in the form of the call chains and therefore have the potential to show better performance in terms of Expense.

We empirically evaluated the proposed approach using 404 real defects from the Defects4J benchmark [41]. Results indicate that except for the two outliers (Chart and Closure) the call-chain-based FL approach can improve the localization effectiveness of 1 to 9 positions (with a relative improvement of 19-48%), compared to Ochiai, a hit-based function-level approach. In the case of defects with ranks worse than 10, this ratio increased even more

(66-98%) on all programs. Furthermore, the defective element could be located in 69% of the cases in the highest-ranked call chains, which turned out to be relatively short on average. Last, but not least, we provide qualitative evidence that, besides improved performances, the proposed approach can provide useful information to the developer performing a debugging task.

The main contributions of this work can be summarized as:

- Based on the high-level concept of function call-chain-based FL presented in [c1], we introduce a new SBFL method that utilizes function call chains instead of individual statements or functions.
- We empirically show that the method can produce about 43% improvement in localization expense compared to simple function-level localization.
- Apart from the final suspiciousness rank, the method offers a set of affected call chains to the user, which can further enhance the localization process. Measurements show that the majority of the defective elements could be located in the highest-ranked call chains.

5.2 Related Work

Similar concepts to function call chains have been explored by other researchers [82, 5], however, not in this detail and not with the FL application in focus. Schröter et al. [84] and Zou et al. [117] suggest assigning a score to the function which is reciprocal to the depth of its first occurrence in the stack trace. They found that this approach was the most successful in localizing the crash faults (which account for about 25% of the defects in Defects4J). Wu et al. [101] extend the call stack with static call graph information and then calculate the suspiciousness scores for the functions.

5.3 Fault Localization on Call Chains

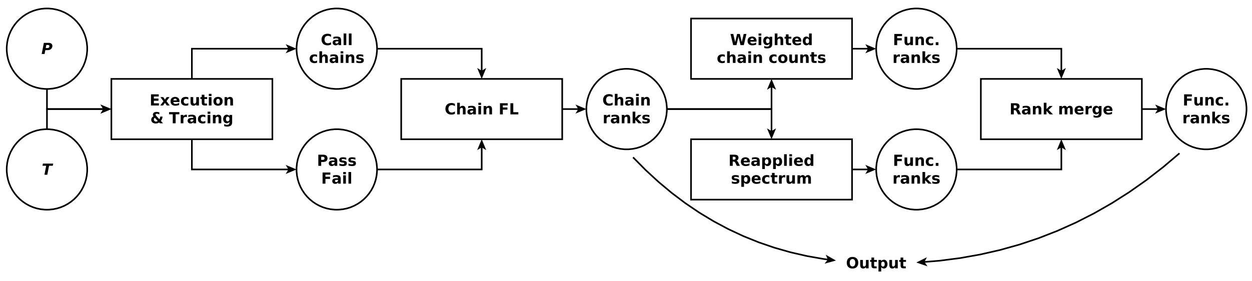


Figure 5.1: *Call-chain-based FL overview*

Figure 5.1 provides a high-level overview of our approach. Using a given set of test cases T, the subject program P is executed while collecting the necessary execution trace information. This is used to produce the function call chains, as well as the test case pass/fail outcomes (more on this in Section 5.3.1). Based on that, we compute the call chain level program spectrum information, which is used to calculate the ranking of the chains according to their suspiciousness levels (discussed in more detail in Section 5.3.2). In the next step, two algorithms are applied to compute the ranking of the functions for FL, which are then merged to produce the final ranking (see Section 5.3.3).

5.3.1 Function Call Chains

Let F be the set of functions in a program P, and T a set of test cases used to test P. Then, a *Call Chain* c is one of the possible deepest *call stack* states occurring during the execution of a test case $t \in T$. Call chains can be efficiently produced from test case executions because only the function entry and exit events need to be recorded and stored in a stack structure. One thing to note here is that the used instrumentor method must be able to handle any non-structured call events such as exceptions and multi-threaded execution. In our method, we collect all distinct call chains occurring during the execution of T, which will be referred to as the call chain set C. We also maintain a set of chains $C(t)$ occurring for each test case t (we say that t *executes* c if $c \in C(t)$). Finally, the set of functions occurring in a chain c will be denoted by $F(c)$.

5.3.2 Call-Chain-Based FL

The first phase of our approach is FL on the call chains. This takes as inputs the test case execution outcomes (pass/fail) and uses a program spectrum representation with the chains as code elements. The output is a ranked list of call chains with the associated suspiciousness scores.

We apply a traditional program spectrum representation based on binary matrices. Let $\mathbf{S}^{ch}$ denote the chain-based spectrum, whose rows represent test cases (elements of T), and columns contain the call chains (elements of C):

$$\mathbf{S}^{ch} = \quad t_i \left\{ \overbrace{\begin{bmatrix} 0/1 & 0/1 & \cdots & 0/1 & 0/1 \\ 0/1 & 0/1 & \cdots & 0/1 & 0/1 \\ & & \ddots & & \\ 0/1 & 0/1 & \cdots & 0/1 & 0/1 \\ 0/1 & 0/1 & \cdots & 0/1 & 0/1 \end{bmatrix}}^{c_j} \right. \quad \mathbf{R}^{ch} = \begin{bmatrix} 0/1 \\ 0/1 \\ \vdots \\ 0/1 \\ 0/1 \end{bmatrix}$$

$\mathbf{S}^{ch}(i, j) = 1$ means that the call chain c_j will occur at least once in the execution of test case t_i. The vector $\mathbf{R}^{ch}$ denotes the test case execution results vector. It is a record of the outcomes of test case runs, namely *pass* (0) or *fail* (1).

For the call chains, any basic SBFL suspiciousness score could be used. In this work, we used the Ochiai score [2], used in recent work [71, 117, 6], and proved to outperform other popular formulae in many situations. To calculate the suspiciousness scores, many formulae rely on some or all of these four basic statistics for each code element c (chain, in our case): $ef(c)$, $nf(c)$, $ep(c)$, and $np(c)$, which count the number of test cases that execute call chain c and fail, do not execute c and fail, execute c and pass, do not execute c and pass, respectively. The Ochiai score does not use $np(c)$ and is computed as:

$$\mathcal{O}(c) = \frac{ef(c)}{\sqrt{(ef(c) + nf(c)) \cdot (ef(c) + ep(c))}} \, .$$

This way, each call chain c will be assigned a suspiciousness score between $[0, 1]$ according to the formula. This, in itself, might be a useful output for the programmers seeking the faulty code element because the high-ranked chains could lead their attention to the faulty element and the context in which it was invoked. However, we proceed to compute also the most suspicious functions, as described in the following.

5.3.3 Locating Functions

A trivial approach for the user to locate the defective function (and statement, respectively) is to consider the highest-ranked call chains and investigate the functions occurring in them (according to our experimentation, this can be successful quite often). But we also propose an approach to produce a ranked list for functions as well based on the call chain scores.

We experimented with various algorithms for this purpose and eventually found out that different strategies may produce good results in different cases. Hence, we decided to use the two best performing strategies and then combine their results, as explained in the following.

5.3.3.1 Weighted Chain Counts

The basic idea with this strategy is to count the number of occurrences of each function in the chains weighted by the respective chain scores from the previous phase. The intuition behind this is that functions frequently occurring in highly ranked chains will be more suspicious. More precisely, for each function $f \in F$ we compute the score $\mathcal{W}$ as:

$$\mathcal{W}(f) = \sum_{c \in C(f)} \mathcal{O}(c) \, , \text{where } C(f) = \{c \mid f \in F(c)\} \, .$$

Note that this score will not fall in the interval $[0, 1]$ which is typical for many other scoring mechanisms. However, this does not affect other parts of the approach since only the relative ranks are subsequently used.

5.3.3.2 Reapplied Spectrum

The second idea for computing function-level scores is to re-apply the spectrum-based approach, but this time on the functions using the call chains in place of the test cases. For this purpose, we treat a call chain as a “proxy” to a test case in the following manner. If its score is greater than a threshold $\mathsf{z} \in [0, 1)$ it is treated as “failing” otherwise as “passing.” Hence, our function-level spectrum has the call chains in its rows and the functions in the columns:

$$\mathbf{S}^{fn} = \quad {}_{c_i}\left\{ \overbrace{\begin{bmatrix} 0/1 & 0/1 & \cdots & 0/1 & 0/1 \\ 0/1 & 0/1 & \cdots & 0/1 & 0/1 \\ & & \ddots & & \\ 0/1 & 0/1 & \cdots & 0/1 & 0/1 \\ 0/1 & 0/1 & \cdots & 0/1 & 0/1 \end{bmatrix}}^{f_j} \right. \quad \mathbf{R}^{fn} = \begin{bmatrix} 0/1 \\ 0/1 \\ \vdots \\ 0/1 \\ 0/1 \end{bmatrix}$$

In this case, a 1 at the matrix position (i, j) means that $f_j \in F(c_i)$, and the entry in the vector $\mathbf{R}^{fn}$ for a chain c_i is 1 if $\mathcal{O}(c_i) > \mathtt{z}$. By adjusting $\mathtt{z}$, one can regulate how "strictly" a suspicious call chain should be considered faulty. We experimented with different thresholds, but in the following, we will set $\mathtt{z} = 0$ as it provided the best results.

The final scores in this case will be computed by re-applying the Ochiai formula to this function-level spectrum, which will be denoted by $\mathcal{R}(f)$ for a function f.

5.3.3.3 Merging the Ranks

The reason for the two ranking methods to behave differently can be traced back to the mentioned coincidental correctness, which – apparently – can affect the chain-based approach as well.

Most SBFL formulae poorly perform when the defective element f has a high $ep(f)$ value compared to $ef(f)$. In the case of our reapplied spectrum technique, this means that f is found in many chains that have $\leq \mathtt{z}$ score, while in fewer chains that have $> \mathtt{z}$ score. This, in turn, can happen if some passing test cases are complex enough to generate a lot of different chains, contrary to the failing ones. We observed that this can happen often in the case of the reapplied spectrum, so in this case, the weighted chain counts technique will perform better because it is not affected by the many passing chains.

Since we do not know in advance which of the two function-level scores will lead to better results in a particular case, so we merged the two ranked lists by alternatively selecting the next element from each of the two lists. The algorithm in Figure 5.2 depicts the approach more precisely.

```
Input R1 {rank according to W scores}
Input R2 {rank according to R scores}
Output {combined rank}
Repeat
  f := next element in R1
  If f is not output yet
    Output f
  f := next element in R2
  If f is not output yet
    Output f
Until R1=empty and R2=empty
```

Figure 5.2: *Rank merging algorithm*

The algorithm has the following property: if the rank position of the faulty element is r_1 in R_1 and r_2 in R_2, in the worst case it will be found in $2 \cdot \min(r_1, r_2)$ steps. This means that if one of the scoring mechanisms is poor compared to the other, the result will depend on the better one. Moreover, if the two ranks are similar, the output will also be similar to them and the mentioned worst case will not be reached. Note that this algorithm does not explicitly handle ties, situations when elements with the same score are ranked subsequently in an arbitrary order. Also, the rank list with which the processing is started is arbitrary. Depending on how these are implemented, the algorithm could produce different final outputs.

Several researchers have used machine learning, such as learning to rank [52], to combine different FL algorithms [105, 6, 117]. We selected to use the above simple approach instead because it guarantees the minimum required rank position and it is practically the same approach the users would follow if they are given two ranked lists to analyze in parallel.

5.4 Empirical Evaluation

The *goal* of the study is to assess the proposed approach based on the combination of call chain scores and function-level SBFL. The *quality focus* is the effectiveness of the approach, compared with state-of-the-art SBFL. The *context* consists of 404 bugs from the Defects4J suite [41].

More specifically, the study aims at addressing the following research questions:

RQ5.1 *What are the properties of the occurring call chains and, in particular, of chains that contain faulty elements?* We measure how often the faulty elements appear in the top-ranked chains. Also, we collect the number and length of the chains because this highly contributes to the usefulness of the chain ranking.

RQ5.2 *How much improvement can the call-chain-based approach achieve compared to basic function-level fault localization?* We measure this property using the fault localization Expense measure (RQ5.2a). At this point, we also compare the two function-level scoring mechanisms of the second phase of our approach and measure how often each of them is better than the other (RQ5.2b).

In the following, we describe the details of the experiment setup and the evaluation methodology.

5.4.1 Study Settings

Our study considered different SBFL formulae *i.e.,* Ochiai, Tarantula, and DStar. Since they provided similar performance in the traditional setting we report only Ochiai results. However, the online appendix [10] includes the results of the other formulae as well.

We performed the experiments on real defects from the Defects4J suite (v1.4.0) [41]. We selected this benchmark because it can be seen as the state of the art in FL research for Java [6, 71, 117, 114, 42, 85], and it includes real defects and programs with non-trivial size and complexity. The dataset provides the fix for each bug as a patch set (called a *version*). Using the patch sets we were able to create change sets that contain data about which functions (Java methods) were affected by each bug fix.

By default, Defects4J utilizes Cobertura [14] to measure code coverage. However, since call chains are needed for our approach we had to use different technology. We used a bytecode instrumentation tool based on Javassist [38] to collect execution traces. This tool uses a compact data structure that was carefully engineered to handle recursive calls and the exceptional amount of data that is generated during the execution of real-life programs.

Unfortunately, some test cases fail if the code is instrumented. These tests assert things that the instrumentation changes *e.g.,* the structure of an object, runtime, contents of the classpath, etc. Since the unexpectedly failing tests would affect the suspiciousness of the covered code elements, we excluded those bugs that include this kind of test. Finally, we considered only those faults for which there is at least one failing and traversing test case. The final set of programs and defects from the Defects4J dataset we used in our experiments is reported in Table 5.1. Numbers regarding size (lines, tests, functions) vary from version to version, here data from the last versions are provided. The last column contains the number of chains generated (also for the last version), which will be explained later.

Table 5.1: *Main Properties of the Defects Used in the Experiments*

Program	**KLOC**	**Tests**	**Bugs**	**Functions**	**Chains**
Chart	96	2 187	25	5 235	41k
Closure	91	7 867	173	8 379	889k
Lang	22	2 270	60	2 353	6k
Math	84	4 371	92	6 351	228k
Mockito	11	1 331	28	1 433	11k
Time	28	4 019	26	3 627	150k
Total	**332**	**22 045**	**404**	**27 378**	**1 325k**

To store the spectrum information matrices and compute the various scores and ranks, we used the SoDA framework [87]. Apart from that only various scripts and spreadsheet editors were used for the calculations.

5.4.2 Measuring the Chain Properties

Compared to a basic function-level SBFL method, the proposed approach requires:

1. To compute the call chains besides simple code coverage information.
2. A larger spectrum matrix, because its columns include different chains rather than functions.
3. An additional step to locate the functions, which is composed of two ranking algorithms and a merging phase. The function-level spectrum requires a matrix whose rows are composed of the different chains which are typically more numerous than the test cases.

To account for these differences, we recorded their basic properties such as the number and size (number of function occurrences) of the chains. The corresponding results are presented in Section 5.5.1.

5.4.3 Evaluation of Fault Localization Effectiveness

To compare our approach to traditional SBFL techniques, we will use the approach presented in Section 2.2.5, *i.e.,* we compute the Expense metric for both approaches and compare them in terms of change relative to traditional SBFL, using both absolute values and relative improvements, and we calculate and present *enabling improvements* as well.

Finally, we compare the rankings achieved by Ochiai with those achieved by the proposed, combined approach. To this aim, a Wilcoxon sign-rank test [16] should normally be used. However, in the context of FL, especially for the easy matches, the test could encounter ties, *e.g.,* when both approaches report the same function as the first. To overcome this limitation, we use, instead, the Wilcoxon-Pratt test [16] which copes with the ties. Since multiple tests are performed (one per program), the p-values have been adjusted using Holm's correction [34]. We complement the Wilcoxon-Pratt test with Cliff's d effect size measure [29].

5.5 Results

In this section, we present the results of our experimental evaluation following our research questions from above.

5.5.1 Properties of Call Chains

As described in Section 5.4.2, we recorded the different properties of data structures during the execution of the experiments and used these to compare our approach to the basic function-level FL. Table 5.1 includes some basic statistics, *i.e.,* the number of functions, tests, and call chains, of the considered programs (their last versions).

The distribution of chain lengths is depicted in the blue boxplots (*i.e.,* the second for each program) of Figure 5.3 (note that outliers are excluded). Although the call chains can be very long (about 3,500 functions), Closure tends to have shorter chains, *i.e.,* about 4 to 26 functions. The average call chain length is 24.8 for the whole dataset and 12.4 if we exclude Closure. (More details can be seen in Column 3 of Table 5.2.)

Table 5.2: *Faulty Elements in Highest-Ranked Chains and Average Chain Lengths*

Program	**Faulty in Highest-Ranked**	**Length**	
		All	**High**
Chart	19 (73%)	8.3	5.7
Closure	98 (56%)	26.0	849.3
Lang	56 (88%)	4.4	5.1
Math	70 (75%)	14.8	13.9
Mockito	20 (69%)	7.8	58.6
Time	21 (78%)	10.1	11.7
Total / Average	**284 (69%)**	**24.8**	**749.2**

We now investigate what is the relationship between the faulty elements and the content of the highly-ranked chains produced in the first phase of our approach. The second column of Table 5.2 shows the number of times (and their ratio) the faulty element can be located in the call chains from the very beginning of the ranked list. In particular, we considered the chains with the highest suspiciousness scores. It is interesting to note that the highest

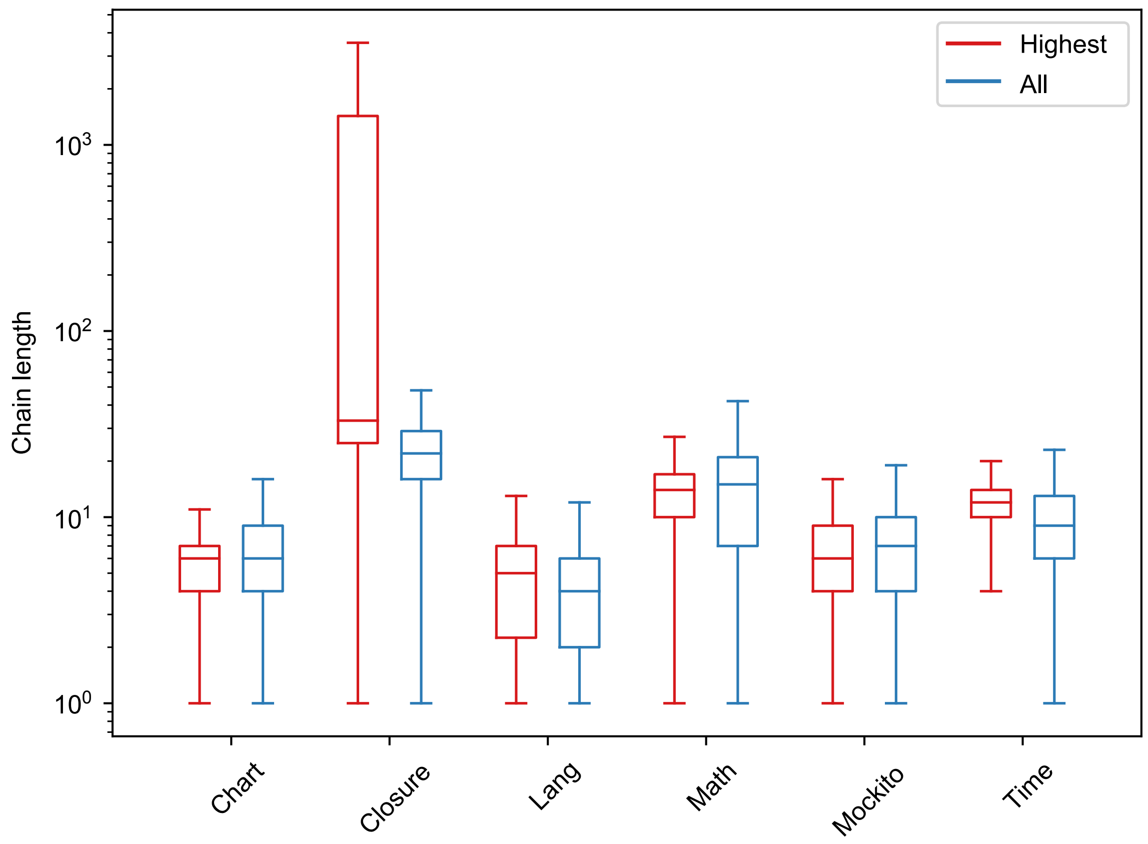


Figure 5.3: *Properties of highest-ranked and all chains*

score was in many cases 1. We can observe from the data that, for all programs, 69% of the defective elements are found in the highest-ranked chains, which is a very high ratio.

It is also interesting to investigate whether these fault-containing chains are any different in terms of their sizes from the general statistics. The red boxes in Figure 5.3 (left-side for each program) depict the length distribution of such chains. As we can observe, the maximum length of these call chains varies from program to program. In general, not considering the outlier Closure, the average length of chains with the highest score is 13.8. Column 4 in Table 5.2 shows the related average values. This finding indicates that the investigation of only the resulting call chains may often lead to finding the fault. However, this process may be supported by the ranked functions in the second phase.

Answer to RQ5.1: 69% of the faulty elements appear in the chains with the highest score values, and these chains contain about 14 functions on average. These two factors contribute to the usefulness of the chain ranks for FL.

5.5.2 Localization Effectiveness

Table 5.3 reports the results for FL effectiveness. Columns “Ochiai” and “Combined” show the absolute and relative Expense values for function-level Ochiai and the proposed approach, respectively. Column “Difference” reports the difference between the average rankings, while column “Relative change” expresses the same as percentage increase/decrease with respect

Table 5.3: *Fault Localization Effectiveness Comparison (Averages Shown)*

Program	**Bugs**	**Ochiai** $E(E')$	**Combined** $E(E')$	**Difference** $E(E')$	**Relative change**	**Ochiai** > 10	**Enabling improvements**	**Relative improvement**
Chart	25	8.3 (0.19%)	10.8 (0.25%)	2.4 (0.06%)	29%	5	2 (8%)	-19.0 (-76%)
Closure	173	99.5 (1.33%)	131.4 (1.77%)	31.9 (0.44%)	32%	106	16 (9%)	-58.8 (-93%)
Lang	60	4.7 (0.23%)	3.5 (0.17%)	-1.1 (-0.05%)	-24%	7	4 (7%)	-15.4 (-66%)
Math	92	11.0 (0.29%)	7.3 (0.19%)	-3.7 (-0.10%)	-34%	27	17 (18%)	-28.1 (-87%)
Mockito	28	25.6 (2.47%)	20.6 (1.98%)	-5.0 (-0.49%)	-19%	9	3 (11%)	-92.0 (-98%)
Time	26	18.3 (0.53%)	9.5 (0.27%)	-8.8 (-0.26%)	-48%	7	2 (8%)	-49.2 (-94%)
Total / Average	**404**	**49.3 (0.89%)**	**61.1 (1.00%)**	**11.9 (0.11%)**	**24%**	**161**	**44 (11%)**	**-43.0 (-91%)**

to Ochiai. Column "Ochiai > 10" reports the number of defects in the programs for which the ranking position is more than 10. "Enabling improvement" indicates how many defects were successfully moved to the 10th or below position by our approach (the percentage is relative to the bug number), and the last column shows the average absolute and relative difference of rankings for such cases.

Table 5.4: *Results of the Wilcoxon-Pratt Test and Cliff's d Effect Size When Comparing Ochiai vs. Combined Approach (Cliff's d is Positive When in Favor of the Combined Approach)*

Program	**Whole evaluation set**			**Bugs ranked** $>$ **10 by Ochiai**		
	p**-value**	d	**Magn.**	p**-value**	d	**Magn.**
Chart	<0.001	-0.05	negligible	0.01	0.12	negligible
Closure	<0.001	-0.27	small	<0.001	-0.01	negligible
Lang	<0.001	0.04	negligible	<0.001	0.59	large
Math	<0.001	0.15	small	<0.001	0.66	large
Mockito	<0.001	-0.25	small	<0.001	0.47	medium
Time	<0.001	-0.04	negligible	<0.001	0.61	large
Overall	**<0.001**	**-0.08**	**negligible**	**<0.001**	**0.14**	**negligible**

For Lang, Math, Mockito, and Time, the improvement is measurable in terms of the Expense metric: this ranges from 1 to about 9 ranking positions on average with a relative change of 19-48%. For Chart and Closure, the proposed algorithm yields ranking positions that are worse by 29-32% on average compared to Ochiai. Note that, the average ranking that Ochiai scores on the bugs of Closure is 99.5, which is already impractical as developers would unlikely investigate such a large number of functions. The reason for this result could be that Closure is different from the other programs in the dataset. It is a JavaScript compiler, which means that it has a very specific code structure and test suite as well. Also note that, despite the poor average performance on Closure, our approach can still deliver enabling improvements in 16 (9%) cases and the improvement is very high -58.8 (-93%) in these cases.

The left-side of Table 5.4 addresses the comparison between the Combined approach and Ochiai from a statistical point of view. Results show that, on the whole evaluation dataset, while the results of the statistical tests show significant differences, the effect size is negligible to small, and in favor of Ochiai.

However, if we look at the results obtained when Ochiai scores a bad ranking position of the correct recommendation, *i.e.,* $> 10th$ (right-side of Tables 5.3 and 5.4), we can observe that 44 of 161 (27%) of defects with ranks higher than 10 could be reduced to below 10 and the average reduction in terms of ranking positions is 43 which is 91% relative improvement. From a statistical point of view, except for Closure, results are in favor of the proposed approach. Also, the effect size is large in three cases (Lang, Math, Time), and medium for Mockito.

Answer to RQ5.2a: In 4 out of 6 cases the call chain-based FL approach could improve the localization effectiveness of Ochiai of 1 to 9 positions on average, with a relative improvement of 19-48%. Also, about 27% of the defects with ranks higher than 10 could be reduced to below 10 with an average reduction of 91%, with statistically significant differences and medium to large effect size.

Our final set of experiments regarding localization effectiveness deals with the two function localization algorithms that work on the ranked chains, which we introduced in Section 5.3.3. As described, the two techniques performed well in different situations, and it was difficult to predict which approach would be better for a particular case. Hence, we follow the described merging approach, which produces an overall better result than the two individually (in each particular case, twice the minimum is guaranteed). Table 5.5 includes the comparison of these two techniques summarized for each program, with the overall average shown in the last row.

Table 5.5: *Comparison of Weighted Chains vs. Reapplied Spectrum (Averages Shown)*

Program	E			**Weighted**	**Reapplied**
	Weighted	**Reapplied**	**Combined**	**better**	**better**
Chart	13.5	10.6	10.8	12	9
Closure	143.6	149.9	131.4	59	112
Lang	3.7	4.0	3.5	23	19
Math	9.3	7.0	7.3	17	52
Mockito	21.0	36.3	20.6	13	15
Time	16.5	8.0	9.5	9	13
Total / Average	**67.5**	**70.1**	**61.1**	**133**	**220**

In columns 2 and 3 of the table, we report the average absolute Expense metrics for the respective techniques, while column 4 includes the same data for the merged outcome. The last two columns include the counts when the respective technique performed better than the other. We can conclude from the data that the combined algorithm indeed is useful because there is a similar number of cases when one of the two rankings is better. We also checked the correlation between the scores produced by the two function-level techniques, and we found that it is close to zero. As expected, the combined approach produced an overall better result than any of the other two, however, both approaches are quite close to the combined one.

Answer to RQ5.2b: When comparing function-level rankings, we found that the reapplied spectrum outperforms the weighted chain counts in more cases (220 vs. 133). It also yields better average scores than the combined approach in some cases, though its overall average is not as low as the scores of the combined rank.

5.5.3 Discussion

Besides the ranking improvement, we argue that the additional information provided by the call chains (stack traces) could help the developer even in situations when the function itself will be further in the rank. As shown in RQ5.1, the faulty element is typically found among the highest-ranked chains. The chains are relatively short (12-13 functions) on average, so investigating the chains themselves in more detail is a good approach during the localization process.

Table 5.6: *Function-Level Ochiai (Base)*

Method	ef	ep	nf	np	Ochiai
forTimeZone	1	6	0	3822	0.3780
getConvertedId	1	6	0	3822	0.3780
getZone	1	131	0	3697	0.0870
getID	1	528	0	3300	0.0435
setDefault	1	3157	0	671	0.0186
getDefault	1	2884	0	944	0.0178

To better illustrate the support provided by call chains during FL, let us consider a real case from our benchmark. Bug number 23 from the Joda-Time Defects4J subject[1] can be located in the method `DateTimeZone.getConvertedId`. This causes one test case, `TestDateTimeZone.testForID_String_old`, to fail. The function-level Ochiai-based FL approach provides the localization scores as shown in Table 5.6. Apart from the mentioned faulty element, all other functions are listed that have a score > 0. It can be seen that all functions are executed by the single failing test case and several passing ones as well. However, two of them are executed by fewer passing tests, *i.e.,* the faulty one and `DateTimeZone.forTimeZone`, which makes them the most suspicious but indistinguishable from each other.

Figure 5.4 shows the relationship of the mentioned functions, which is an excerpt of a call graph belonging to this program. `DateTimeZone.forTimeZone` is the main function called by the test case, which apart from the faulty `DateTimeZone.getConvertedId` calls `ZoneInfoProvider.getZone` as well. The other directly called functions are setup and teardown helper functions for the test case. The reason the base algorithm cannot distinguish between `forTimeZone` and `getConvertedId` is that the latter is always called (both from failing and passing test cases) by the former, and no additional information is available to the algorithm.

By introducing the concept of the call chains, `forTimeZone` $\rightarrow$ `getConvertedId` can be investigated separately along with all other call chains. In particular, `forTimeZone` $\rightarrow$ `getZone` is interesting as it also relates to the suspicious `forTimeZone` but represents a

[1]https://github.com/JodaOrg/joda-time/commit/14dedcb

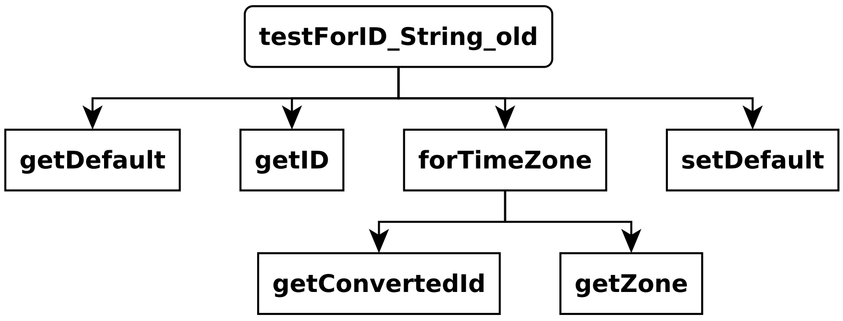


Figure 5.4: *Call graph of `TestDateTimeZone.testForID_String_old`*

Table 5.7: *Chain-Level Ochiai*

Chain	**ef**	**ep**	**nf**	**np**	**Ochiai**
forTimeZone→getZone	1	2	0	3826	0.5774
forTimeZone→getConvertedId	1	2	0	3826	0.5774
getID	1	9	0	3819	0.3162
setDefault	1	2882	0	946	0.0186
getDefault	1	2887	0	941	0.0186

different context. Table 5.7 presents the localization scores calculated by the first phase of the proposed approach, namely the suspiciousness scores for the chains. Similarly, we only show those chains that have > 0 scores. We can observe that apart from the two mentioned chains, the other (one-element) chains are represented as well because they are also part of the failing test run (and also of some passing runs as well).

Table 5.8: *Function-Level Ochiai (Reapplied)*

Chain	**ef**	**ep**	**nf**	**np**	**Ochiai**
getConvertedId	1	4	4	87692	0.2000
forTimeZone	2	50	3	87646	0.1240
setDefault	1	78	4	87618	0.0503
getZone	1	77	4	87619	0.0506
getID	1	376	4	87320	0.0230

Again, the two highest-ranked chains cannot be distinguished from each other because both are executed in the same situations by the failing and passing test cases. However, the next phase of our approach can pinpoint the faulty elements, because it combines the information about suspicious chains with the functions they contain. Namely, in the reapplied spectrum technique, we treat all suspicious call chains as "failing" and by counting their frequency for each function and the frequency of non-suspicious chains for the same, we can select the most suspicious function. Table 5.8 shows the statistics for this phase. As can be seen, the highest score is given to `getConvertedId`, followed by `forTimeZone`. The explanation for this can also be seen in the corresponding numbers used by the Ochiai formula. Although `forTimeZone` can be found in more suspicious chains than `getConvertedId` (2 vs.

1) it is found in much more non-suspicious chains as well (50 as opposed to 4). `forTimeZone` is a common method called by many test cases, passing and failing, and present in many different chains, but its specific branching to the faulty `getConvertedId` is less frequent and is typical to the failing test case.

This example is realistic and shows one possible benefit of the approach. However, we had to limit its complexity to be able to clearly explain it. The ranking positions 1 and 2, used in the example, are equally good in practical situations, but in more complex cases, the context provided by the call chains could be much more useful.

In summary, the two outputs produced by our approach (*i.e.,* the ranked list of most suspicious call chains in the first phase and the merged ranked list of functions in the second) can be used in different scenarios — to be empirically evaluated in future work through user studies — to complement hit-based approaches like Ochiai. In the first scenario, the user can start localizing the fault by observing the ranked chains. If the fault is located this way, the context of the investigated chains also informs about the possible ways to fix the defect. If there are many high-ranked chains with equally high scores, the user can rely on the final result of the ranked functions from the second phase, and focus on those functions only. In the second scenario, the user starts from the ranked list of functions from the second phase, and if the defect is not easily found, looks at the highest-ranked call chains (and the functions with high ranks in them) for clues about the possible contexts leading to the failed test cases.

5.5.4 Threats to Validity

Concerning *construct validity*, we relied on a widely used measurement for SBFL, *i.e.,* the Expense metric. The likelihood of errors in the dataset is limited, as the Defect4J suite is widely used in research, and we carefully reviewed the code for call chain extraction.

The main threat to the *internal validity* of this study is that, as explained in Section 5.4.1, we used 404 defects out of a total of 438 that were available in the Defect4J version we used, because we could not compute call chains for all of them. However, the selection was not based on the performances of the proposed algorithm and of Ochiai, and the comparison was in any case performed on the same dataset. Since this limitation was mainly technological due to the nature of the programs in the dataset we used, it will not affect use cases when one could apply the approach in a real usage scenario.

Conclusion validity of this study is supported by the use of appropriate statistical procedures, namely a non-parametric test suitable to deal with ties (Wilcoxon-Pratt test), the use of Holm's correction to avoid fishing the error rate, and Cliff's d effect size.

Concerning *external validity*, while the evaluation of the study has been performed on 404 real defects from Java programs, it is still desirable to replicate the work on a larger and possibly more diverse dataset.

5.6 Conclusions

This thesis investigates the use of *function call chains* for Spectrum-Based Fault Localization (SBFL). Call chains are instances of call stack traces, and these are useful artifacts occurring at runtime which can often help developers in debugging.

Results indicate that except for the two outliers the proposed approach can achieve a significant improvement in terms of the FL expense, about 19-48%, which is even higher in the case of worse ranking positions (over 10).

The highest-ranked call chains provide useful information for a better understanding of the context of the defect (in 69% of the cases, the defective element is in the highest-ranked chains), and could even provide hints for the fixation of the bug. For instance, the call chain indicates which function invokes the defective function when the fault manifests in a failure.

In future work, we plan to conduct user studies to evaluate the practical usefulness of call chains, following scenarios as the ones described in Section 5.5.3. In addition, it could be interesting to evaluate our approach on more benchmarks *e.g.,* Bugs.jar [83], BugsJS [30], etc. The rank combination approach we used could be replaced by a more sophisticated approach taking into account other properties of the spectrum, or by the learning-to-rank model.

Thesis III:

1. I worked on the development of the theoretical background of applying the concept of function call chains in SBFL.
2. I designed and implemented a bytecode instrumentation tool that was used to collect the call chains in the experiments.
3. I implemented the call-chain-based FL approach, including the weighted chain count, the reapplied spectrum, and the rank merging algorithms.
4. I performed experiments using real faults from the Defects4J benchmark to evaluate this approach.
5. I measured and analyzed the effectiveness and efficiency of this method.

Response to challenges. This thesis responds to challenge **C3** by introducing a new approach which utilizes the additional contextual information of the collected function call-chains to successfully improve the effectiveness and efficiency of SBFL.

Publications. The concept of using call chains for fault localization, the corresponding FL algorithm, the experiments and the results were published at the *International Conference on Software Analysis, Evolution and Reengineering* (SANER'20) [c1]. As a follow up of this work, a new algorithm which also utilizes call chains was published at the *International Conference on Software Analysis, Evolution and Reengineering* (SANER'21) [c4]. However, contrary to this thesis, that approach uses call chains indirectly *i.e.,* to compute a count-based spectrum that replaces the traditional hit-based one. In addition, an extension of the latter work was published in the *Special Issue on Intelligent Bug Fixing of Journal of Software: Evolution and Process* (JSEP'22) [j3].

Appendices

A Comparison of Coverage Measurement Tools

A.1 Source-Code-Based Tools

In this appendix, we present additional results of comparing the results of the two methods employing source code instrumentation, Clover and Test Coverage.

As these two tools generate instrumented source code, it is possible to compare their instrumentation algorithms on a high level by investigating the instrumented code itself. To do so, we manually checked the instrumented sources and investigated the probe points, *i.e.,* the locations where extra code were injected into the original source code. We found that the two tools identified and instrumented exactly the same source code elements. The main technical difference between the two tools is that while Test Coverage uses boolean vectors to store coverage data, Clover has a complex mechanism for calculating which part of the production code is exercised (this enables per-test coverage measurement as well). Thus, in general, Test Coverage inserts less extra code into the original source. Another difference is the handling of such code which does not have a conventional form of a Java method but will be included in the bytecode as a special method (*e.g.,* static initialization code of the class or anonymous methods). Both tools recognize and instrument these parts of the source code, but Clover reports them as methods, while Test Coverage includes them in the class coverage only.

In the next step, we calculated the raw overall coverages for our subject systems with these two tools in order to see how much their results differ. Table A.1 shows the associated results. Columns 2 and 3 show the overall coverage ratios as produced by the tools, while the last column includes the percentage difference. The numbers in the last row represent the averages of the absolute differences.

We can observe that the aforementioned differences of the tools cause small differences in the overall coverage results. Note, that we used Clover$^{\text{glob}}$ which is the value measured globally, including cross-submodule coverage for submodule-based systems, and this is how Test Coverage works too. Unfortunately, we were not able to produce per-test coverage values using Test Coverage, as this would have required the individual execution of the test

Table A.1: *Comparison of the Overall Coverages Computed by the Source Code Tools (Clover and Test Coverage)*

Program	**Clover$^{\text{glob}}$**	**Test Coverage**	**Clover$^{\text{glob}}$ vs. Test Coverage**
Checkstyle	93.82%	93.77%	-0.05%
Lang	93.28%	93.13%	-0.15%
Math	84.65%	85.59%	+0.94%
Time	89.94%	90.94%	+1.00%
MapDB	76.06%	78.58%	+2.52%
Netty	46.66%	48.93%	+2.27%
OrientDB	39.84%	39.92%	+0.08%
Oryx	27.51%	27.68%	+0.17%
Average Difference			**0.90%**

cases and consequently large-scale modifications in the build environments. Hence, we could not perform such a comparison of the tools.

A.2 Bytecode-Based Tools

In this appendix, we present additional results of comparing the results of the three tools employing bytecode instrumentation: JaCoCo, JCov, and Cobertura.

We calculated the raw overall coverages for our subject systems with these tools in order to see how much their results differ. Table A.2 shows the associated results. Columns 2–4 are the overall coverage ratios as produced by the tools "out of the box" (*i.e.,* without any modifications made to them[1], only the necessary parameters have been set). The last three columns show the pairwise differences in the percentage, while the numbers in the last row represent the averages of the absolute differences.

Quantitatively, the differences between these tools were at most 4%, and the behavior of the tools was different on the different subjects. We made some deeper but basically still quantitative analysis: we compared the per-test coverage results of the tools. Unfortunately, we were not able to produce such results using Cobertura, so we compared JaCoCo and JCov in this respect.

In Figure A.1 we present the differences in test case coverage vectors. For each test case, we use a coverage vector in which each element corresponds to a single code element. We compared such vector pairs for JaCoCo and JCov for each test case using the Hamming distance measure and normalized the result by the length of the vectors. Figure A.1 shows the corresponding data in form of histograms.

For the first four subject programs, data shows that most of the vector pairs are the same and the difference is less than 1% for the others. For MapDB and Netty, there are very few vector pairs that match exactly, but most of them are still close to each other. In the case of Oryx and OrientDB, about half of the vector pairs match exactly, the difference in the majority of the cases is less than 1%, but there are differences as high as 5% or even 14%.

[1]There was one exception to this: in Cobertura, we disabled the feature of skipping the analysis of generated source code, as this was not implemented in the other two tools.

Table A.2: *Comparison of the Overall Coverages Computed by the Bytecode Tools (Cobertura, JaCoCo and JCov)*

Program	**JaCoCo**	**JCov**	**Cobertura**	**JCov vs. JaCoCo**	**Cobertura vs. JaCoCo**	**Cobertura vs. JCov**
Checkstyle	53.85%	52.20%	54.79%	-1.65%	0.94%	2.59%
Lang	93.29%	92.81%	94.08%	-0.49%	0.79%	1.28%
Math	85.59%	87.41%	88.78%	1.82%	3.19%	1.37%
Time	91.36%	91.33%	91.55%	-0.03%	0.19%	0.22%
MapDB	79.65%	79.12%	78.46%	-0.53%	-1.20%	-0.67%
Netty	47.41%	49.10%	45.51%	1.69%	-1.90%	-3.59%
OrientDB	38.40%	42.02%	42.23%	3.62%	3.83%	0.21%
Oryx	29.62%	26.33%	25.60%	-3.29%	-4.03%	-0.74%
Average Difference				**1.64%**	**2.01%**	**1.33%**

After the manual investigations, most of these high differences are found to be tolerable outliers.

Similarly to the previous set of experiments, we performed a per-method comparison of the coverages. Here the vectors were assigned to code elements and a vector element corresponded to a test case. Figure A.2 shows the differences in these vectors for JaCoCo and JCov in form of histograms.

We got the same results as in the case of test case vectors for the first 4 subject programs, namely, most of the vector pairs are the same and the rest of them differ by at most 1%. Here, the last 4 programs are similar to each other: a lower part of the vector pairs matches exactly, but there are some higher differences as well. Later, these high differences turned out to be explainable outliers.

To find the cause of the differences (especially of the high Hamming distances) we observed from these experiments, we investigated the detailed coverage of the three tools manually as well. We concluded that the main cause of the differences was mostly due to the slightly different handling of compiler-generated methods and nested classes in the bytecode (such as the methods generated for nested classes). Since the overall quantitative differences were at most 4% and they were concerned mostly generated methods, which are less important for code coverage analysis, and the high individual Hamming distances could also be traced back to these methods, we concluded that one representative tool of the three should be sufficient for further experiments.

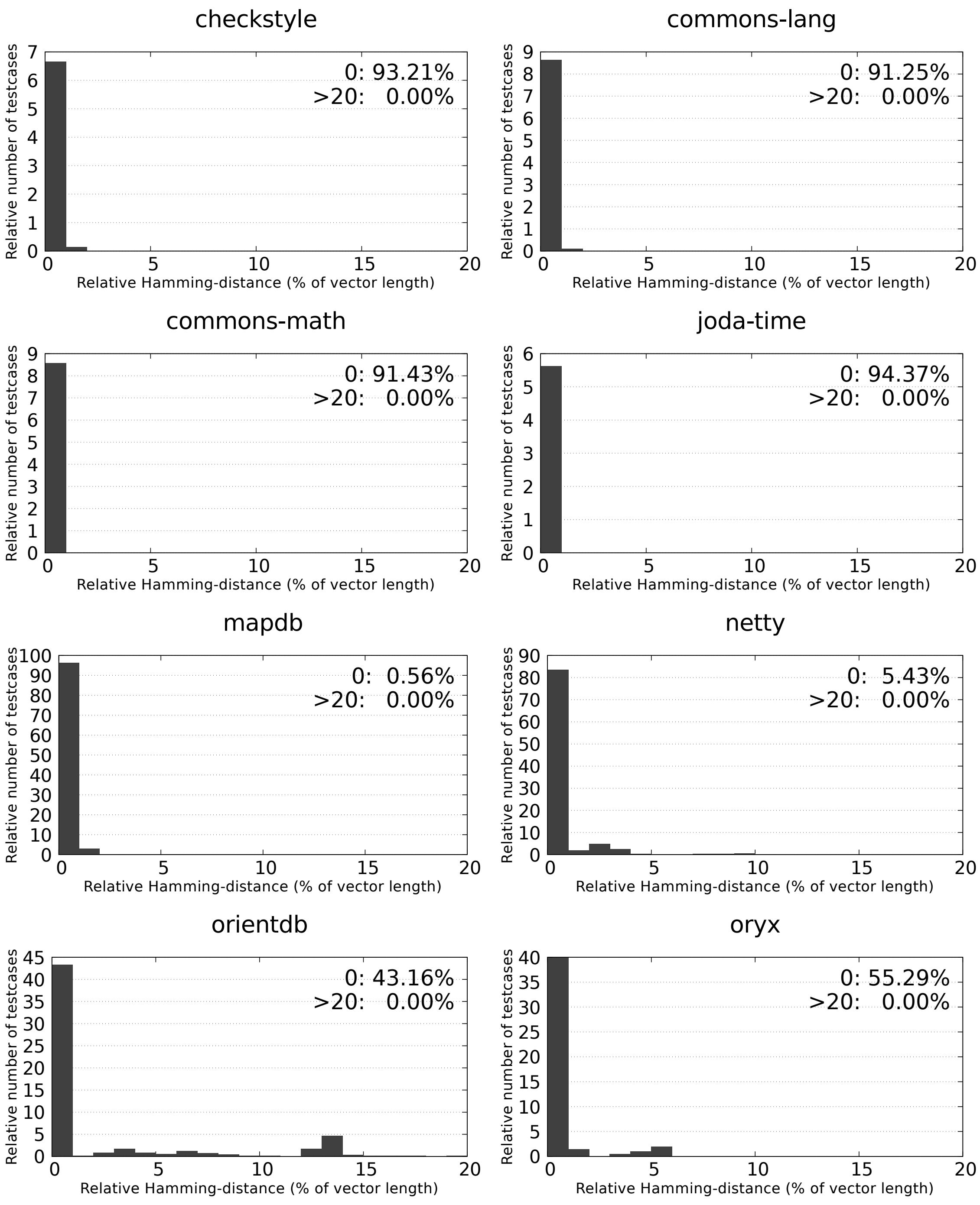


Figure A.1: *Relative Hamming distances of test case vectors (JaCoCo vs. JCov)*

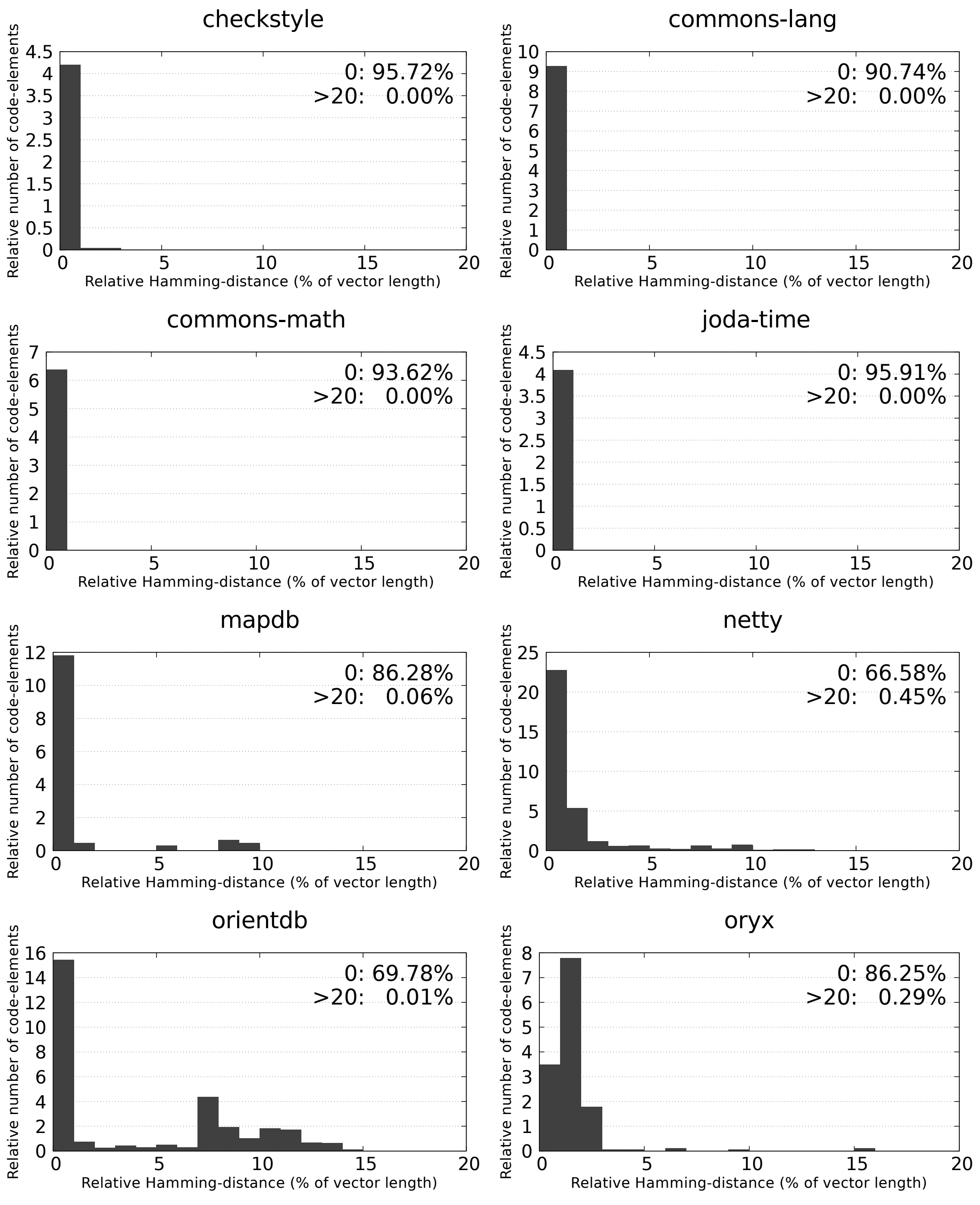


Figure A.2: *Relative Hamming distances of code element vectors (JaCoCo vs. JCov)*

B Summary

Code coverage measurement plays an important role in white-box testing in industrial practice and academic research as well. Moreover, several areas are highly dependent on code coverage, including test case generation, test prioritization, and fault localization, among others. Out of these areas, this dissertation focuses on two main topics, and the thesis points are divided into two parts accordingly. The first part consists of one thesis point that discusses the differences between methods for measuring code coverage in Java and the effects of these differences. The second part is composed of two thesis points on fault localization, more specifically, improving the efficiency of spectrum-based approaches by incorporating external information, *e.g.,* users' knowledge, or context data extracted from call chains. The relation between these thesis points and their supporting publications is shown in Table B.1.

I Code Coverage Measurement

1 Effects of Measurement Methods on Java Code Coverage and Their Impact on Applications

The contributions of this thesis point – related to code coverage measurement methods, and the impact of measurement discrepancies on test prioritization and test suite reduction – are discussed in Chapter 3.

Software testers have long established the theory and practice of code coverage measurement: various types of coverage criteria like statement, branch, and others [12], as well as technical solutions including various kinds of instrumentation methods [106]. This work was motivated by our experience in using code coverage measurement tools for the Java programming language. Even in a relatively simple setting (a method-level analysis of medium size software with popular and stable tools), we found significant differences in the outputs of different tools applied for the same task. The differences in the computed coverages might have serious impacts on different applications, such as false confidence in white-box testing, difficulties in coverage-driven test case generation, and inefficient test prioritization, just to name a few.

Various reasons might exist for such differences and surely there are certain issues that tool builders have to face, but we have found that in the Java environment, the most notable issue is how *code instrumentation* is done. The code instrumentation technique is used to place “probes” into the program, which will be activated upon runtime to collect the necessary information about code coverage. In Java, there are two fundamentally different instrumentation approaches: *source code* level and *bytecode* level. Both approaches have benefits and drawbacks, but many researchers and practitioners prefer to use bytecode instrumentation due to its various technical benefits [106]. However, in most cases the application of code coverage is on the source code, hence it is worthwhile to investigate and compare the two approaches.

This work reports on an empirical study to compare the code coverage results provided by tools using the different instrumentation types for Java coverage measurement on the method level. We initially considered a relatively large set of candidate tools referenced in literature and used by practitioners, and then we started the experiments with five popular tools which seemed mature enough and actively used and developed. Overall coverage results are compared using these tools, but eventually, we selected one representative for each instrumentation approach to perform the in-depth analysis of the differences (JaCoCo and Clover). The measurements are made on a set of 8 benchmark programs from the open-source domain which are actively developed real-size systems with large test suites. The differences are systematically investigated both quantitatively (how much the outputs differ) and qualitatively (what the causes for the differences are). Not only do we compare the coverages directly, but investigate the impact on a possible application of coverage measurement in more detail as well. The chosen applications are test prioritization and test suite reduction based on code coverage information.

The majority of earlier work on the topic dealt with lower-level analyses such as statements and branches. Instead, we performed experiments on the granularity of Java methods in real-size Java systems with realistic test suites. We found that – contrary to our preliminary expectations – even at this level there might be significant differences between bytecode instrumentation and source code instrumentation approaches. Method level granularity is often the viable solution due to the large system size. Furthermore, if we can demonstrate the weaknesses of the tools at this level, they are expected to be present at the lower levels of granularity as well.

We found that the overall coverage differences between the tools can vary in both directions, and in the case of seven out of the eight subject programs they are at most 1.5%. However, for the last program, we measured an extremely large difference of 40% (this was then attributed to the different handling of generated code).

We looked at more detailed differences as well with respect to individual test cases and program elements. In many applications of code coverage (in debugging, for instance) subtle differences at this level may lead to serious confusion. We measured differences of up to 14% between the individual test cases, and differences of over 20% between the methods. In a different analysis of the results, we found that a substantial portion of the methods in the subjects was affected by this inaccuracy (up to 30% of the methods in one of the subject programs).

We systematically investigated the reasons for the differences and found that some of them were tool-specific, while the others would be attributed to the instrumentation approach. This list of reasons may be used as a guideline for the users of coverage tools on how to avoid or workaround the issues when a bytecode instrumentation-based approach is used.

We also measured the effect of the differences on the application of code coverage to test prioritization. We found that the prioritized lists produced by the tools differed significantly (with correlations below 0.5), which means that the impact of the inaccuracies might be significant. We think that this low correlation is a great risk: in other words, it is not possible to predict the potential amplification of a given coverage inaccuracy in a particular application. This also affects any related research which is based on bytecode instrumentation coverage measurement to a large extent.

The Author's Contributions

The author of this thesis worked on the overview of theoretical differences in code coverage measurement tools for Java. He took part in the collection, categorization, testing, and selection of code coverage measurement tools. After establishing the measurement environment, he also took part in the collection, configuration, and selection of Java programs on which the experiments were executed. He measured and analyzed the differences in code coverage of Java bytecode and source code instrumentation tools. The author worked on the systematic investigation of discrepancies in coverage data and their causes, and helped develop fixes and recommendations for the correction of the issues. He performed experiments to analyze the effects of the found differences on coverage-based applications, namely test selection, and test prioritization.

The publications related to this thesis point are:

- [c3] Dávid Tengeri, Ferenc Horváth, Árpád Beszédes, Tamás Gergely, and Tibor Gyimóthy. "Negative effects of bytecode instrumentation on Java source code coverage". In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER).* vol. 1. IEEE. 2016, pp. 225–235
- [j2] Ferenc Horváth, Tamás Gergely, Árpád Beszédes, Dávid Tengeri, Gergő Balogh, and Tibor Gyimóthy. "Code coverage differences of Java bytecode and source code instrumentation tools". In: *Software Quality Journal* 27.1 (Mar. 2019), pp. 79–123

II Fault Localization

1 Interactive Fault Localization

The contributions of this thesis point – related to interactive fault localization – are discussed in Chapter 4.

Recent studies highlighted some barriers to the wide adoption of the SBFL methods, including a high number of suggested elements to investigate [102, 70], applicability of theoretical results in practice [48], little experimental results with real faults [71], validity issues of empirical research [90], and so on. With this work, we aim at bringing closer the applicability of SBFL methods to practice by involving users' knowledge to the process.

The basic intuition behind SBFL is that code elements (statements, blocks, functions, etc.) that are exercised by comparably more failing test cases than passing ones are more suspicious to contain a fault. Suspiciousness is usually expressed by assigning one value to each code element (the *suspiciousness score*), which can then be used to *rank* the code elements. When this ranked list is given to the developer for investigation, it is hoped that the fault will be found near the beginning of the list. Studies revealed that the number of elements that have to be investigated before finding the fault is crucial to the adoption of the method in practice. In particular, research showed that if the faulty element is beyond the 5th element (or 10th according to other studies), the method will not be used by practitioners because they need to investigate too many elements [71, 102, 70, 47]. A further problem is that there are no guarantees that any scoring mechanism will show sufficiently good correlation between the score and the actual faults [97, 71, 103, 111]. One additional reason an SBFL method may fail is that these approaches provide only the ranked list of code elements, however this gives little or no information about the context of bugs which makes their comprehension a cumbersome task for developers.

It seems that automatic SBFL methods require external information – not just the program spectra and test case outcomes – to improve on state-of-the-art performance and be more suitable in practical settings. In this work, we propose a form of an *Interactive Fault Localization* approach, called *iFL*. In traditional SBFL, the developer has to investigate several locations before finding the faulty code elements, and all the knowledge they a priori have or acquire during this process is not fed back into the SBFL tool. In our approach, the developer interacts with the fault localization algorithm by giving feedback on the elements of the prioritized list.

We build on our and other researchers' observations, intuitions and experiences, and we hypothesize that a programmer, when presented with a particular code element, in general has a strong intuition whether any other elements belonging to the same containing higher level code entity should be considered in fault localization. With this intuition, developers can also make a decision ("judge") about the code snippets associated with the item they are currently examining. This allows them to narrow down the search space (*i.e.,* set of the suspicious code elements) more efficiently, which could speed up finding the bug. For example, when users go through the ranked list of suspicious methods, in addition to the examined code element, they could have knowledge about its class, which information can be "fed back" into *iFL* to modify the suspiciousness value of other methods in that class or even exclude items to be examined. This way, larger code parts can be repositioned in their suspiciousness in the hope to reach the faulty element earlier.

We evaluated the approach in two sets of experiments. First, we used *simulation* to predict the effect of interactivity. We simulated user actions during hypothetical fault finding in well-known bug benchmarks, and measured the Expense metric improvements with respect to the following traditional SBFL formulae: Tarantula [40], Ochiai [2], and DStar [99]. We relied on two benchmarks: artificial defects from the SIR repository [17] and real defects from Defects4J [41]. Results show that the method can significantly improve the fault localization efficiency: in both benchmarks, for 32-57% of the faults their ranking position is reduced from beyond the 10th position to between the 1-10th position. Taking into account all the defects, the localization efficiency in terms of Expense improved on average by 71-79%. For reference, we implemented a closely related interactive FL algorithm proposed by Gong et al. [26], called Talk, in our simulation framework. We compared the performance of *iFL* to Talk on the real faults from Defects4J, and found that *iFL* has a significant advantage over Talk. We also modelled user imperfection, which was rarely studied in related interactive SBFL research. We addressed this aspect from two viewpoints: the user's knowledge and confidence. Experiments simulating these two factors show that *iFL* can outperform a traditional non-interactive SBFL method notably even at low user confidence and knowledge levels.

In the second stage, we performed a quantitative evaluation of the successfulness of *iFL* usage by *real users*. We invited students and professional programmers to solve a set of fault localization tasks using the implementation of the *iFL* approach in a controlled experiment. The goal was to find out whether using the tool shows actual benefits in terms of finding more bugs or finding them more quickly, and this also showed promising results. This experiment also helped us better understand the developers' thought processes and the weaknesses of the approach, and gave us possible directions for future enhancements.

The Author's Contributions

The author worked on the development of the theoretical background of applying the concept of interactive feedback in fault localization. He also worked on the development of the theoretical background of applying the concept of user imperfection factors (confidence and knowledge) in the evaluation of fault localization. Following the theoretical design, the author implemented the simulation framework as a basis for testing interactive fault localization approaches. The implementation of the *iFL* approach in the simulation framework is also the author's own work. He performed experiments using seeded and real faults from the SIR and Defects4J benchmark to evaluate *iFL*. He measured and analyzed the effectiveness and efficiency of *iFL* in the aforementioned simulated environment. The author took part in the reimplementation of the TALK algorithm, and the execution of the subsequent comparative experiments and analyses. He contributed to the design and development of the iFL4Eclipse plug-in which implements *iFL* in the Eclipse IDE. The author was involved in the design, execution, and evaluation of the user studies.

The publications related to this thesis point are:

- [j1] Ferenc Horváth, Árpád Beszédes, Béla Vancsics, Gergo Balogh, László Vidács, and Tibor Gyimóthy. "Using contextual knowledge in interactive fault localization". In: *Empirical Software Engineering* 27 (Aug. 2022)
- [c2] Ferenc Horváth, Árpád Beszédes, Béla Vancsics, Gergő Balogh, László Vidács, and Tibor Gyimóthy. "Experiments with Interactive Fault Localization Using Simulated and Real Users". In: *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2020, pp. 290–300
- [w1] Ferenc Horváth, Victor Schnepper Lacerda, Árpád Beszédes, László Vidács, and Tibor Gyimóthy. "A New Interactive Fault Localization Method with Context Aware User Feedback". In: *2019 IEEE 1st International Workshop on Intelligent Bug Fixing (IBF)*. Feb. 2019, pp. 23–28
- [t1] Gergő Balogh, Victor Schnepper Lacerda, Ferenc Horváth, and Árpád Beszédes. *iFL for Eclipse – A Tool to Support Interactive Fault Localization in Eclipse IDE*. Presented in the Tool Demo Track of the 12th IEEE International Conference on Software Testing, Verification and Validation (ICST'19). Apr. 2019
- [p1] Gergő Balogh, Ferenc Horváth, and Árpád Beszédes. "Poster: Aiding Java Developers with Interactive Fault Localization in Eclipse IDE". in: *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. 2019, pp. 371–374

2 Call-Chain-Based Fault Localization

The contributions of this thesis point – related to call-chain-based fault localization – are discussed in Chapter 5.

The state-of-the-art approach to SBFL is to use the so-called "hit-based" spectra [32] with statements as basic code elements. Researchers proposed many different scoring mechanisms, but these are essentially all based on counts of passing/failing and traversing/non-traversing test cases in different combinations [97, 71, 103]. Popular suspiciousness scores are Tarantula [40], Ochiai [2], and DStar [99], among others.

One reason why an SBFL formula may fail is what is referred to as *coincidental correctness* [94, 60, 9]. This is the situation when a test case traverses a faulty element without failing. This can happen quite often since not all exercised elements may have an impact on the computation performed by a test case [61], and if there are relatively more such cases than traversing and failing ones, the suspiciousness score will be negatively affected [60].

Based on the vast amount of research performed in the field, it seems that variations to these basic approaches may yield only marginal improvements, and that perhaps some more radical changes in how we approach the problem are required in order to achieve more significant gains. For example, by combining conceptually different approaches [117], or by involving additional information to the process. Early attempts to incorporate control or data flow information, for instance [77, 32], have not been further developed because it soon became apparent that they are difficult to scale to large programs and real defects.

Beszédes et al. [c1] propose the concept of enhancing traditional SBFL with *function call chains* on which the FL is performed. Function call chains are snapshots of the call stack occurring during execution and as such can provide valuable context to the fault being traced. Call chains (and call stack traces) are artifacts that occur during program execution and are well-known to programmers who perform debugging and can show, for instance, that a function may fail if called from one place and perform successfully when called from another. There is empirical evidence that stack traces help developers fix bugs Schröter et al. [84], and Zou et al. [117] showed that stack traces can be used to locate *crash-faults*.

In this work, based on the high-level concept of function call-chain-based FL presented in [c1], we propose a novel SBFL algorithm, that computes ranking on all occurring call chains during execution, and then selects the suspicious functions from these ranked chains using a function-level (*i.e.,* method-level for object-oriented languages like Java) spectrum-based algorithm, Ochiai in particular [2].

Our approach works at a higher granularity than statement-level approaches (previous work suggests that function-level is a suitable granularity for the users [6, 117]). At the same time, we provide more context in the form of the call chains and therefore have the potential to show better performance in terms of Expense.

We empirically evaluated the proposed approach using 404 real defects from the Defects4J benchmark [41]. Results indicate that except for the two outliers (Chart and Closure) the call-chain-based FL approach can improve the localization effectiveness of 1 to 9 positions (with a relative improvement of 19-48%), compared to Ochiai, a hit-based function-level approach. In the case of defects with ranks worse than 10, this ratio increased even more (66-98%) on all programs. Furthermore, the defective element could be located in 69% of the cases in the highest-ranked call chains, which turned out to be relatively short on average. Last, but not least, we provide qualitative evidence that, besides improved performances, the proposed approach can provide useful information to the developer performing a debugging task.

The Author's Contributions

The author worked on the development of the theoretical background of applying the concept of function call chains in SBFL. He designed and implemented a bytecode instrumentation tool that was used to collect the call chains in the experiments. Following the theoretical design, the author implemented the call-chain-based FL approach, including the weighted chain count, the reapplied spectrum, and the rank merging algorithms. He performed experiments using real faults from the Defects4J benchmark to evaluate this approach, as well as, he measured and analyzed the effectiveness and efficiency of this method.

The publications related to this thesis point are:

- ♦ [c1] Árpád Beszédes, Ferenc Horváth, Massimiliano Di Penta, and Tibor Gyimóthy. "Leveraging contextual information from function call chains to improve fault localization". In: *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. 2020, pp. 468–479
- ♦ [c4] Béla Vancsics, Ferenc Horváth, Attila Szatmári, and Árpád Beszédes. "Call Frequency-Based Fault Localization". In: *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2021, pp. 365–376
- ♦ [j3] Béla Vancsics, Ferenc Horváth, Attila Szatmári, and Árpád Beszédes. "Fault localization using function call frequencies". In: *Journal of Systems and Software* 193 (2022), p. 111429. ISSN: 0164-1212

Table B.1 summarizes the main publications and how they relate to the thesis points.

№	[c3]	[j2]	[j1]	[c2]	[w1]	[t1]	[p1]	[c1]	[c4]	[j3]
I.	♦	♦								
II.			♦	♦	♦	♦	♦			
III.								♦	♦	♦

Table B.1: *Thesis contributions and supporting publications*

C
Összefoglaló

A kódlefedettség mérés fontos szerepet játszik a white-box tesztelésben az ipari gyakorlatban és a tudományos kutatásokban egyaránt. Ráadásul számos terület nagymértékben függ a kódlefedettségtől, többek között a tesztesetek generálása, a tesztek priorizálása és a hibalokalizáció is. Ezen területek közül jelen disszertáció két fő témára fókuszál, és a tézispontok ennek megfelelően két részre oszlanak. Az első rész egy tézispontból áll, amely a kódlefedettség mérésére szolgáló módszerek közötti különbségeket és ezen különbségek hatásait tárgyalja. A második rész két tézispontból áll, amelyek a hibalokalizációval foglalkoznak, pontosabban a spektrumalapú megközelítések hatékonyságának javításával külső információk, például a felhasználók tudása vagy a hívási-láncokból kinyert kontextusadatok bevonásával. A tézispontok és az azokat alátámasztó publikációk közötti kapcsolatot a C.1 táblázat foglalja össze.

I Forráskód lefedettség mérés

1 A mérési módszerek hatásai a kódlefedettségre és ezek hatása az alkalmazásokra

Ezt a tézispontot, amely a mérési módszereknek a kódlefedettségre, valamint az alkalmazásokra gyakorolt hatásairól szól, a 4. fejezet tárgyalja bővebben.

A szoftvertesztelők már rég kidolgozták a kódlefedettség mérés elméletének és gyakorlatának alapjait: a különböző típusú lefedettségi kritériumokat, mint például az utasítás, az elágazás, stb [12], valamint a technikai megoldásokat, beleértve a különböző mérési módszereket [106]. Ezt a munkát a Java programmozási nyelvre vonatkozó kódlefedettség mérésére szolgáló eszközök használatával kapcsolatos tapasztalataink motiválták. Még egy viszonylag egyszerű környezetben is, mint a metódus-szintű elemzés közepes méretű, népszerű, és stabil eszközökkel, jelentős különbségeket találtunk a kimenetekben az egyazon feladatra alkalmazott különböző eszközök eredményei között. A kiszámított lefedettségek közötti különbségek komoly hatással lehetnek a különböző alkalmazásokra, például hamis bizalmat kelthetnek a white-box tesztelés irányában, nehézségeket indukálhatnak a lefedettségvezérelt teszteset generálásban, vagy negatívan befolyásolhatják a tesztek priorizálásának hatékonyságát.

Az ilyen különbségeknek számos oka lehet, és akadnak bizonyos problémák, amelyekkel az eszközkészítőknek szembe kell nézniük, de úgy találtuk, hogy a Java környezetben a legjelentősebb probléma a kód instrumentálásának módja. A kódinstrumentálási technikával "szondákat" helyeznek el a programban, amelyek futtatáskor aktiválódnak, hogy összegyűjtsék a szükséges információkat a kódlefedettségről. Java-ban két alapvetően eltérő instrumentálási megközelítés létezik: a forráskód szintű és a bájtkód szintű. Mindkét megközelítésnek vannak előnyei és hátrányai, de sok kutató és gyakorlati szakember a bájtkód instrumentálást részesíti előnyben annak különböző technikai előnyei miatt [106]. A legtöbb esetben a kódlefedettség alkalmazása a forráskódra vezetendő vissza, ezért érdemes megvizsgálni és összehasonlítani a két megközelítést.

Ez a munka egy olyan empirikus vizsgálatról számol be, amely a Java kódlefedettség mérésére szolgáló különböző típusú eszközök által szolgáltatott kódlefedettségi eredményeket hasonlítja össze. Kezdetben a szakirodalomban hivatkozott és a gyakorlati szakemberek által használt eszközök viszonylag nagy halmazát vettük figyelembe, majd a kísérleteket öt olyan népszerű eszközzel kezdtük, amelyek elég kiforrotnak tűntek, és amelyeket aktívan használnak és fejlesztenek. Az általános lefedettségi eredményeket ezekkel az eszközökkel hasonlítottuk össze, de végül minden egyes instrumentálási megközelítésből kiválasztottunk egy-egy reprezentatív eszközt a különbségek mélyreható elemzéséhez (JaCoCo és Clover). A méréseket 8 olyan nyílt forráskódú programon végeztük el, amelyek aktívan fejlesztett, valós méretű, nagy tesztkészletekkel rendelkező rendszerek. A különbségeket szisztematikusan vizsgáljuk mind mennyiségi (mennyire különböznek a kimenetek), mind minőségi (mi a különbségek oka) szempontból. Nemcsak közvetlenül hasonlítjuk össze a lefedettségeket, hanem részletesebben is vizsgáljuk a lefedettségmérés alkalmazásaira (teszt priorizálás és teszt szelekció) gyakorolt hatását.

A témával kapcsolatos korábbi munkák többsége alacsonyabb szintű elemzésekkel, például utasításokkal és elágazásokkal foglalkozott. Ehelyett metódus szintre vonatkozó kísérleteket végeztünk valós méretű Java-rendszerekben, reális tesztkészletekkel. Azt találtuk, hogy – előzetes várakozásainkkal ellentétben – még ezen a szinten is jelentős különbségek lehetnek a bájtkód instrumentáló és a forráskód instrumentáló megközelítések között. A metódus szintű granularitás a nagy rendszerméret miatt gyakran a megvalósítható megoldás. Továbbá, ha ezen a szinten ki tudjuk mutatni az eszközök gyengeségeit, akkor azok várhatóan az alacsonyabb szinteken is jelen lesznek.

Megállapítottuk, hogy az eszközök közötti általános lefedettségi különbségek mindkét irányban változhatnak, és a nyolc vizsgált programból hét esetében legfeljebb 1,5%-osak. Az utolsó program esetében azonban rendkívül nagy, 40%-os különbséget mértünk (ezt a későbbi vizsgálatok során a generált kódrészletek eltérő kezelésének tulajdonítottuk). Részletesebb különbségeket is vizsgáltunk az egyes tesztesetek és programelemek tekintetében. A kódlefedettség számos alkalmazásánál (például a hibakeresésnél) az ilyen szintű finom különbségek komoly zavart okozhatnak. Az egyes tesztesetek között akár 14%-os, a módszerek között pedig több mint 20%-os különbségeket mértünk. Az eredmények további elemzése során azt találtuk, hogy a programok metódusainak jelentős részét érintette ez a pontatlanság (az egyik programban a módszerek közel 30%-át).

Szisztematikusan megvizsgáltuk a különbségek okait, és azt találtuk, hogy a különbségek egy része eszköz-specifikus, míg a többi a mérési megközelítésnek tulajdonítható. A felfedett okok listája iránymutatásként szolgálhat a lefedettségi eszközök felhasználói számára, hogy miként kerüljék vagy hárítsák el a problémákat, ha bájtkód-alapú megközelítést alkalmaznak.

Azt is megmértük, hogy a különbségek milyen hatással vannak a kódlefedettség-alapú teszt priorizálásra. Azt találtuk, hogy az eszközök által előállított priorizált listák jelentősen eltértek egymástól (0,5 alatti korrelációkkal), ami azt jelenti, hogy a pontatlanságok hatása jelentős lehet. Úgy gondoljuk, hogy ez az alacsony korreláció nagy kockázatot jelent. Más szóval, nem lehet megjósolni egy adott lefedettségi pontatlanság potenciális felerősödését egy adott alkalmazásban. Ez természetesen minden olyan kapcsolódó kutatást is érinthet, amely nagymértékben támaszkodik a bájtkód-alapú lefedettség mérésre.

A szerző hozzájárulása

Jelen disszertáció szerzője a Java kódlefedettség mérő eszközök elméleti különbségeinek áttekintésén dolgozott. Részt vett a kódlefedettséget mérő eszközök összegyűjtésében, kategorizálásban, tesztelésben és kiválasztásában. A mérési környezet kialakítása után részt vett a kísérletekhez használt Java programok összegyűjtésében, konfigurálásában és kiválasztásában is. Mérte és elemezte a Java bájtkód és a forráskód lefedettség mérő eszközök különbségeit. A szerző dolgozott a lefedettségi adatok eltéréseinek és azok okainak szisztematikus vizsgálatán, és részt vett a hibák kijavítására szolgáló javítások és ajánlások kidolgozásában. Kísérleteket végzett a talált eltéréseknek a lefedettségen alapuló alkalmazásokra, nevezetesen a teszt szelekcióra és a teszt priorizálásra gyakorolt hatásának elemzése céljából.

A tézispont a következő publikációkra épül:

- [c3] Dávid Tengeri, Ferenc Horváth, Árpád Beszédes, Tamás Gergely és Tibor Gyimóthy. “Negative effects of bytecode instrumentation on Java source code coverage”. *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER).* 1. köt. IEEE. 2016, 225–235. old.
- [j2] Ferenc Horváth, Tamás Gergely, Árpád Beszédes, Dávid Tengeri, Gergő Balogh és Tibor Gyimóthy. “Code coverage differences of Java bytecode and source code instrumentation tools”. *Software Quality Journal* 27.1 (2019. márc.), 79–123. old.

II Hibalokalizáció

2 Interaktív hibalokalizáció

Ezt a tézispontot, amely az interaktív hibalokalizációhoz kapcsolódik, a 4. fejezet tárgyalja bővebben.

A közelmúltban készült tanulmányok rávilágítottak a spektrum alapú hibalokalizációs módszerek (továbbiakban: SBFL) széleskörű alkalmazásának néhány akadályára. Ilyen akadály, többek között a vizsgálandó elemek nagy száma [102, 70], az elméleti eredmények gyakorlatban történő alkalmazhatósága [48], a kevés kísérleti eredmény valós hibákkal [71], valamint az empirikus kutatások érvényességi problémái [90]. Célunk, hogy a felhasználók tudásának bevonásával közelebb hozzuk az SBFL módszerek alkalmazhatóságát a gyakorlathoz.

Az SBFL alapvetése a következő: azon kódelemek (utasítások, blokkok, függvények stb.) esetében, amelyeket viszonylag sok sikertelen és kevés sikeres teszteset fed le, ott nagyobb az esélye annak, hogy az adott elemek hibát tartalmaznak. A gyanússágot általában úgy fejezzük ki, hogy minden kódelemhez egy valós számot rendelünk (ún. *gyanússági érték*), amely aztán a kódelemek rangsorolására használható. Amikor ezt a rangsorolt listát átadjuk a fejlesztőknek, hogy vizsgálják meg az elemeket, akkor remélhetőleg a hiba a lista elején lesz majd található. Kutatások kimutatták, hogy a tényleges hiba megtalálása előtt megvizsgálandó elemek száma döntő fontosságú ezen módszer gyakorlati alkalmazása szempontjából. Megállapításra került, hogy ha a hibás elem az 5. helynél (vagy más tanulmányok szerint a 10. helynél) hátrébb van a listában, akkor a módszert a felhasználók nem fogják használni, mert túl sok elemet kell megvizsgálniuk [71, 102, 70, 47]. További probléma, hogy nincs garancia arra, hogy bármelyik számítási mechanizmus kellően jó korrelációt mutat a gyanússági érték és a tényleges hibák között [97, 71, 103, 111]. Az SBFL módszer sikertelenségének egyik további oka, hogy ezek a megközelítések csak a kódelemek rangsorolt listáját adják meg, ez azonban kevés vagy semmilyen információt nem ad a hibák kontextusáról, ami a fejlesztők számára nehézkes feladattá teszi a hibák megértését.

Úgy tűnik, hogy az automatikus SBFL módszereknek valamilyen külső információra van szükségük – a kódlefedettségen és a tesztesetek eredményén felül – ahhoz, hogy lehetőség nyíljon a manapság népszerű megközelítések teljesítményének javítására, illetve, hogy alkalmasabbak legyenek a gyakorlati környezetben történő használatra. Ebben a tézisben az *interaktív hibalokalizációs* megközelítés egy formáját terjesztjük elő, amelyet *iFL*-nek nevezünk. A hagyományos SBFL-ben a fejlesztőnek több elemet kell megvizsgálnia, mielőtt megtalálja a hibás kódelemeket, és az összes tudást, amellyel eredendően rendelkezik vagy amelyet e folyamat során szerez, azt elveszítjük, nincs mód rá, hogy visszatápláljuk ezt az információt az SBFL eszközbe. A mi megközelítésünkben a fejlesztő interakcióba léphet a hibalokalizációs algoritmussal azáltal, hogy visszajelzést ad a priorizált lista elemeiről.

A saját és más kutatók megfigyeléseire, intuícióira és tapasztalataira építünk, és feltételezzük, hogy a programozónak általában – amikor egy adott kódelemmel találkozik – erős intuíciója van arról, hogy az azonos, vagy magasabb szintű kódegységhez tartozó más elemeket figyelembe kell-e venni a hibalokalizáció során. Ennek az intuíciónak a segítségével a fejlesztők az éppen vizsgált elemhez tartozó kódrészletekről is tudnak döntést hozni. Ez lehetővé teszi számukra, hogy hatékonyabban szűkítsék a keresési teret (azaz a gyanús kódelemek halmazát), ami felgyorsíthatja a hiba megtalálását. Például, amikor a felhasználók a rangsorolt listán haladnak végig, a vizsgált kódelemen túl az osztályáról is rendelkezhetnek ismeretekkel, amely információ "visszatáplálható" az *iFL*-be, ezáltal módosítható az adott osztályba tartozó további elemek gyanússági értéke, vagy akár teljesen ki is zárhatóak bizonyos vizsgálandó elemek. Ily módon nagyobb kódrészletek is áthelyezhetőek a listában abból a célból, hogy hamarabb elérjük a hibás elemet.

A megközelítést két kísérletsorozatban értékeltük ki. Az első fázisban *szimulációt* használtunk az interaktivitás hatásának előrejelzésére, megbecsülésére. Szimuláltuk a hibakeresés során végzett felhasználói műveleteket és mértük az *Expense* metrika[1] javulását a következő hagyományos SBFL megközelítésekhez képest: Tarantula [40], Ochiai [2], és DStar [99]. Két adathalmazra támaszkodtunk: mesterséges hibákra a SIR-ből [17] és valódi hibákra a Defects4J-ből [41]. Az eredmények azt mutatják, hogy a módszer jelentősen javítja a hibalokalizáció hatékonyságát: mindkét adathalmazon a 10. pozíciót meghaladó pozícióról az 1-10. pozíció közé csökkent a hibás elemek pozíciója a hibák 32-57%-ban. Az összes hibát figyelembe véve pedig átlagosan 71-79%-kal javult a hatékonyság. Összevetés céljából a Gong és tsai. [26] által publikált, Talk nevű, interaktív hibalokalizációs algoritmust is újra implementáltuk a szimulációs keretrendszerünkben. Összemértük az *iFL* és a Talk teljesítményét a Defects4J valós hibáin, és azt találtuk, hogy az *iFL* jelentős előnyben van a Talk-kal

[1]A hiba megtalálásához szükséges lépések számát fejezi ki.

szemben. A felhasználói "bizonytalanságot" is modelleztük, amelyet a kapcsolódó kutatásokban ritkán vizsgáltak eddig. Ezt két szemszögből vizsgáltuk meg: a felhasználó tudása és megabiztossága. A két tényezőt szimuláló kísérletek azt mutatják, hogy az *iFL* még alacsony felhasználói magabiztosság és tudásszint esetén is képes felülmúlni a hagyományos, nem interaktív SBFL módszert.

A második szakaszban kvantitatív kiértékelést végeztünk az *iFL* használatának sikerességéről *valós felhasználók* által. Diákokat és hivatásos programozókat kértünk fel arra, hogy egy ellenőrzött kísérletben hibalokalizációs feladatokat oldjanak meg az *iFL* megközelítés egy implementációjának felhasználásával. A cél annak az elemzése volt, hogy az eszköz használata valóban előnyös-e, azaz segítségével több hiba és kevesebb idő alatt található-e meg – ebben a kísérletben is ígéretes eredmények születtek. Ez a kísérlet segített továbbá jobban megérteni a módszerünk gyengeségeit és a fejlesztők gondolatmenetét, valamint sikerült általa meghatároznunk lehetséges irányokat a jövőbeli továbbfejlesztéshez.

A szerző hozzájárulása

A disszertáció szerzője részt vett az interaktív visszacsatolás koncepciójának a hibalokalizációban, továbbá a felhasználói bizonytalansági tényezők koncepciójának a hibalokalizáció értékelésében való alkalmazásának elméleti hátterének kidolgozásában. Ezt követően megvalósította a szimulációs keretrendszert, amely az interaktív hibalokalizációs megközelítések tesztelésének alapjául szolgált. Az *iFL* megközelítés megvalósítása a szimulációs keretrendszerben szintén a szerző saját munkája. A módszer kiértékelésének során a SIR és a Defects4J adathalmazokból származó injektált és valós hibák felhasználásával végzett kísérleteket. Mérte és elemezte az *iFL* hatékonyságát a szimulációs környezet segítségével. Részt vett a Talk algoritmus újra implementálásában, valamint az ezt követő összehasonlító kísérletek és elemzések végrehajtásában. Közreműködött az iFL4Eclipse tervezésében és fejlesztésében, amely az *iFL*-t a népszerű Eclipse fejlesztőkörnyezetben valósítja meg. Valamint részt vett a felhasználói tanulmányok tervezésében, végrehajtásában és kiértékelésében.

A tézispont a következő publikációkra épül:

- ♦ [j1] Ferenc Horváth, Árpád Beszédes, Béla Vancsics, Gergo Balogh, László Vidács és Tibor Gyimóthy. "Using contextual knowledge in interactive fault localization". *Empirical Software Engineering* 27 (2022. aug.)
- ♦ [c2] Ferenc Horváth, Árpád Beszédes, Béla Vancsics, Gergő Balogh, László Vidács és Tibor Gyimóthy. "Experiments with Interactive Fault Localization Using Simulated and Real Users". *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2020, 290–300. old.
- ♦ [w1] Ferenc Horváth, Victor Schnepper Lacerda, Árpád Beszédes, László Vidács és Tibor Gyimóthy. "A New Interactive Fault Localization Method with Context Aware User Feedback". *2019 IEEE 1st International Workshop on Intelligent Bug Fixing (IBF)*. 2019. febr., 23–28. old.
- ♦ [t1] Gergő Balogh, Victor Schnepper Lacerda, Ferenc Horváth és Árpád Beszédes. *iFL for Eclipse – A Tool to Support Interactive Fault Localization in Eclipse IDE*. Presented in the Tool Demo Track of the 12th IEEE International Conference on Software Testing, Verification and Validation (ICST'19). 2019. ápr.
- ♦ [p1] Gergő Balogh, Ferenc Horváth és Árpád Beszédes. "Poster: Aiding Java Developers with Interactive Fault Localization in Eclipse IDE". *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. 2019, 371–374. old.

3 Hívási-lánc alapú hibalokalizáció

Ezt a tézispontot, amely a hívási-lánc alapú hibalokalizációhoz kapcsolódik, az 5. fejezet tárgyalja bővebben.

Az SBFL legnépszerűbb megközelítése az úgynevezett bináris spektrumokra [32] épül és a legszéleskörűbben használt elemzési szint az utasítás vagy sor szint. A kutatók számos különböző statisztikai megközelítést javasoltak már, de ezek alapvetően mind a különböző kombinációkban előforduló sikeres/bukó és fedő/nem fedő tesztesetek számolásán alapulnak [97, 71, 103]. Többek között ilyen népszerű módszerek közé tartoznak a Tarantula [40], az Ochiai [2] és a DStar [99] is.

Az egyik ok, amiért egy SBFL formula rosszul működhet, az úgynevezett *véletlenszerű helyesség* [94, 60, 9], azaz az a jelenség, amikor egy teszteset egy valójában hibás elemen úgy halad át, hogy a teszt kimenetele sikeres lesz. Ez sajnos elég gyakran előfordulhat, mivel a lefedett elemnek nem minden esetben van hatásuk a teszteset által elvégzett számításokra és így a teszt végső kimenetelére [61], sőt, ha viszonylag sok ilyen eset fordul elő és az adott elemeket fedő bukó tesztesetek száma kicsi, akkor ez a gyanússági értéket negatívan befolyásolhatja [60].

A területen végzett temérdek kutatás alapján úgy tűnik, hogy ezen alapvető megközelítések kombinálása csak elhanyagolható javulást eredményezhet, illetve, hogy talán radikálisabb változtatásokra van szükség a probléma megközelítésének módjában ahhoz, hogy jelentősen jobb eredményeket érjünk el. Például alapvetően különböző megközelítések kombinálásával [117], vagy további információknak a folyamatba való bevonásával. A vezérlési vagy adatáramlási információk bevonására tett korai kísérletek, például [77, 32], nem kerültek továbbfejlesztésre, mert hamar kiderült, hogy nehezen skálázhatók nagy programokra és valós hibákra.

Ebben a munkában a hagyományos SBFL továbbfejlesztését tűztük ki célul a *(függvény)-hívási-láncok* koncepciójának felhasználásával. A hívási-láncok a végrehajtás során előforduló hívási verem pillanatfelvételei, és mint ilyenek, értékes információt biztosítanak a éppen vizsgálandó hiba kontextusáról. A hívási-láncok a program végrehajtása során előforduló, a hibakeresést végző programozók által jól ismert források, amelyek megmutatják például, hogy egy függvény hibásan működik, ha egy adott helyről hívják meg, ellenben hibátlanul működik, ha egy másik helyről hívják. Empirikus bizonyítékok vannak arra, hogy a *stack trace*-ek segítenek a fejlesztőknek a hibák kijavításában [84], valamint Zou és tsai. [117] kimutatták, hogy a *stack trace*-ek felhasználhatók program összeomlást okozó hibák felkutatására.

Konkrétabban, egy SBFL algoritmust javasolunk, amely a végrehajtás során előforduló összes hívási-lánc kiszámolja a gyanússági értékeket, majd az ezek alapján rangsorolt láncokból választja ki a gyanús függvényeket egy függvényszintű (objektumorientált nyelvek, például Java esetén metódusszintű) spektrum alapú algoritmus segítségével.

Habár a megközelítésünk durvább granularitást használ, mint az utasításszintű megközelítések (korábbi munkák szerint a függvényszint megfelelő granularitás a felhasználók számára [6, 117]), ugyanakkor a hívásláncok formájában több információt biztosítunk a kontextusról, így potenciálisan jobb teljesítményt érhetünk el.

A javasolt megközelítést a Defects4J adathalmaz [41] 404 valós hibájának felhasználásával értékeltük ki. Az eredmények azt mutatják, hogy a két kiugró eset (Chart és Closure) kivételével a hívási-lánc alapú megközelítés 1-9 pozícióval (19-48%-os relatív javulással) képes javítani a lokalizációs hatékonyságot a hagyományos Ochiai megközelítéshez képest. A 10-

nél rosszabb rangú hibák esetében ez az arány még jelentősebb: átlagosan 66-98% az összes vizsgált hibára vetítve. Továbbá, a hibás elemet az esetek 69%-ában lehetett megtalálni a legmagasabb rangú (leggyanúsabb) hívási-láncokban, amelyek átlagosan viszonylag rövidnek bizonyultak. Végül, de nem utolsósorban, bizonyítékot szolgáltatunk arra, hogy a jobb teljesítmény mellett a javasolt megközelítés hasznos információkkal szolgálhat a hibakeresési feladatot végző fejlesztő számára.

A szerző hozzájárulása

A szerző egy olyan kontextus alapú hibalokalizációs módszer elméleti hátterének kidolgozásában vett részt, amely a hibalokalizáció javításához a függvényhívási láncok koncepcióját használja fel. Megtervezte és megvalósította azt a bájtkód instrumentációs eszközt, amelyet a kísérletekben a hívási-láncok kinyerésére használt. A tervezést követően a szerző megvalósította a hívási-lánc alapú hibalokalizációs megközelítést, beleértve a súlyozott lánc számlálásos, az újraalkalmazott spektrum és a rangsorolt lista összevonó algoritmusokat. Kísérleteket végzett a Defects4J adathalmaz valós hibáinak felhasználásával a megközelítés kiértékelésének céljából, valamint mérte és elemezte a módszer hatékonyságát és eredményességét.

A tézispont a következő publikációkra épül:

- ♦ [c1] Árpád Beszédes, Ferenc Horváth, Massimiliano Di Penta és Tibor Gyimóthy. “Leveraging contextual information from function call chains to improve fault localization”. *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. 2020, 468–479. old.
- ♦ [c4] Béla Vancsics, Ferenc Horváth, Attila Szatmári és Árpád Beszédes. “Call Frequency-Based Fault Localization”. *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2021, 365–376. old.
- ♦ [j3] Béla Vancsics, Ferenc Horváth, Attila Szatmári és Árpád Beszédes. “Fault localization using function call frequencies”. *Journal of Systems and Software* 193 (2022), 111429. old. ISSN: 0164-1212

A tézispontokat és a kapcsolódó publikációkat a C.1. táblázat összegzi.

№	[c3]	[j2]	[j1]	[c2]	[w1]	[t1]	[p1]	[c1]	[c4]	[j3]
I.	♦	♦								
II.			♦	♦	♦	♦	♦			
III.								♦	♦	♦

C.1. táblázat. *A tézispontokhoz kapcsolódó publikációk*

Acknowledgments

Firstly, I would like to thank Árpád Beszédes, Ph.D., my supervisor, for his professional help and guidance during my Ph.D. studies. Secondly, I am also grateful for all of my co-authors, with whom we achieved goals that sometimes seemed unachievable. Finally, I would also like to express my gratitude for the continuous support of my beloved family.

The research was supported by the European Union project RRF-2.3.1-21-2022-00004 within the framework of the Artificial Intelligence National Laboratory and by project TKP2021-NVA-09 implemented with the support provided by the Ministry of Innovation and Technology of Hungary from the National Research, Development and Innovation Fund, financed under the TKP2021-NVA funding scheme.

The Author's Publications on the Subjects of the Theses

Journal Papers

[j1] Ferenc Horváth, Árpád Beszédes, Béla Vancsics, Gergo Balogh, László Vidács, and Tibor Gyimóthy. "Using contextual knowledge in interactive fault localization". In: *Empirical Software Engineering* 27 (Aug. 2022).

[j2] Ferenc Horváth, Tamás Gergely, Árpád Beszédes, Dávid Tengeri, Gergő Balogh, and Tibor Gyimóthy. "Code coverage differences of Java bytecode and source code instrumentation tools". In: *Software Quality Journal* 27.1 (Mar. 2019), pp. 79–123.

[j3] Béla Vancsics, Ferenc Horváth, Attila Szatmári, and Árpád Beszédes. "Fault localization using function call frequencies". In: *Journal of Systems and Software* 193 (2022), p. 111429. ISSN: 0164-1212.

Conference Papers

[c1] Árpád Beszédes, Ferenc Horváth, Massimiliano Di Penta, and Tibor Gyimóthy. "Leveraging contextual information from function call chains to improve fault localization". In: *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. 2020, pp. 468–479.

[c2] Ferenc Horváth, Árpád Beszédes, Béla Vancsics, Gergő Balogh, László Vidács, and Tibor Gyimóthy. "Experiments with Interactive Fault Localization Using Simulated and Real Users". In: *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2020, pp. 290–300.

[c3] Dávid Tengeri, Ferenc Horváth, Árpád Beszédes, Tamás Gergely, and Tibor Gyimóthy. "Negative effects of bytecode instrumentation on Java source code coverage". In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Vol. 1. IEEE. 2016, pp. 225–235.

[c4] Béla Vancsics, Ferenc Horváth, Attila Szatmári, and Árpád Beszédes. "Call Frequency-Based Fault Localization". In: *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2021, pp. 365–376.

Workshop Papers

[w1] Ferenc Horváth, Victor Schnepper Lacerda, Árpád Beszédes, László Vidács, and Tibor Gyimóthy. "A New Interactive Fault Localization Method with Context Aware User Feedback". In: *2019 IEEE 1st International Workshop on Intelligent Bug Fixing (IBF)*. Feb. 2019, pp. 23–28.

Poster Papers

[p1] Gergő Balogh, Ferenc Horváth, and Árpád Beszédes. "Poster: Aiding Java Developers with Interactive Fault Localization in Eclipse IDE". In: *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. 2019, pp. 371–374.

Tool Papers

[t1] Gergő Balogh, Victor Schnepper Lacerda, Ferenc Horváth, and Árpád Beszédes. *iFL for Eclipse – A Tool to Support Interactive Fault Localization in Eclipse IDE*. Presented in the Tool Demo Track of the 12th IEEE International Conference on Software Testing, Verification and Validation (ICST'19). Apr. 2019.

The Author’s Further Related Publications

[f1] Tamás Gergely, Gergő Balogh, Ferenc Horváth, Béla Vancsics, Árpád Beszédes, and Tibor Gyimóthy. “Analysis of Static and Dynamic Test-to-code Traceability Information”. In: *Acta Cybernetica* 23.3 (Jan. 2018), pp. 903–919.

[f2] Tamás Gergely, Gergő Balogh, Ferenc Horváth, Béla Vancsics, Árpád Beszédes, and Tibor Gyimóthy. “Differences between a static and a dynamic test-to-code traceability recovery method”. In: *Software Quality Journal* 27.2 (June 2019), pp. 797–822.

[f3] Ferenc Horváth, Szabolcs Bognár, Tamás Gergely, Róbert Rácz, Árpad Beszédes, and Vladimir Marinkovic. “Code Coverage Measurement Framework for Android Devices”. In: *Acta Cybern.* 21.3 (Aug. 2014), pp. 439–458. ISSN: 0324-721X.

[f4] Ferenc Horváth and Tamás Gergely. “Structural information aided automated test method for magic 4GL”. In: *Acta Cybernetica* 22.1 (Jan. 2015), pp. 81–99.

[f5] Ferenc Horváth, Béla Vancsics, László Vidács, Árpád Beszédes, Dávid Tengeri, Tamás Gergely, and Tibor Gyimóthy. “Test suite evaluation using code coverage based metrics”. English. In: *CEUR Workshop Proceedings*. Vol. 1525. CEUR-WS, 2015, pp. 46–60.

[f6] András Kicsi, Viktor Csuvik, László Vidács, Ferenc Horváth, Árpád Beszédes, Tibor Gyimóthy, and Ferenc Kocsis. “Feature analysis using information retrieval, community detection and structural analysis methods in product line adoption”. In: *Journal of Systems and Software* 155 (2019), pp. 70–90. ISSN: 0164-1212.

[f7] András Kicsi, László Vidács, Viktor Csuvik, Ferenc Horváth, Árpád Beszédes, and Ferenc Kocsis. “Supporting Product Line Adoption by Combining Syntactic and Textual Feature Extraction”. In: *New Opportunities for Software Reuse*. Ed. by Rafael Capilla, Barbara Gallina, and Carlos Cetina. Cham: Springer International Publishing, 2018, pp. 148–163.

[f8] László Vidács, Ferenc Horváth, József Mihalicza, Béla Vancsics, and Árpád Beszédes. “Supporting software product line testing by optimizing code configuration coverage”. In: *2015 IEEE eighth international conference on software testing, verification and validation workshops (ICSTW)*. IEEE. 2015, pp. 1–7.

[f9] László Vidács, Ferenc Horváth, Dávid Tengeri, and Árpád Beszédes. “Assessing the test suite of a large system based on code coverage, efficiency and uniqueness”. In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Vol. 2. IEEE. 2016, pp. 13–16.

References

[1] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. “On the Accuracy of Spectrum-based Fault Localization”. In: *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*. 2007, pp. 89–98. ISBN: 0-7695-2984-4.

[2] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan J. C. van Gemund. “A Practical Evaluation of Spectrum-based Fault Localization”. In: *J. Syst. Softw.* 82.11 (Nov. 2009), pp. 1780–1792. ISSN: 0164-1212.

[3] H. Agrawal, R. A. De Millo, and E. H. Spafford. “An execution-backtracking approach to debugging”. In: *IEEE Software* 8.3 (May 1991), pp. 21–26. ISSN: 0740-7459. DOI: `10.1109/52.88940`.

[4] Khalid Alemerien and Kenneth Magel. “Examining the Effectiveness of Testing Coverage Tools: An Empirical Study.” In: *International Journal of Software Engineering and Its Applications* 8.5 (2014), pp. 139–162.

[5] Glenn Ammons, Thomas Ball, and James R. Larus. “Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling”. In: *SIGPLAN Not.* 32 (1997), 85–96. ISSN: 0362-1340. DOI: `10.1145/258916.258924`. URL: `http://doi.acm.org/10.1145/258916.258924`.

[6] Tien-Duy B Le, David Lo, Claire Le Goues, and Lars Grunske. “A learning-to-rank based fault localization approach using likely invariants”. In: *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM. 2016, pp. 177–188.

[7] A. Bandyopadhyay and S. Ghosh. “Proximity based weighting of test cases to improve spectrum based fault localization”. In: *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. Nov. 2011, pp. 420–423.

[8] A. Bandyopadhyay and S. Ghosh. “Tester Feedback Driven Fault Localization”. In: *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. Apr. 2012, pp. 41–50.

[9] Benoit Baudry, Franck Fleurey, and Yves Le Traon. “Improving test suites for efficient fault localization”. In: *28th international conference on Software engineering*. ICSE ’06. Shanghai, China: ACM, 2006, pp. 82–91. ISBN: 1-59593-375-1.

[10] Árpád Beszédes, Ferenc Horváth, Massimiliano Di Penta, and Tibor Gyimóthy. *Leveraging Contextual Information from Function Call Chains to Improve Fault Localization - Online Appendix*. `https://chainfl.github.io`.

[11] Walter Binder, Jarle Hulaas, and Philippe Moret. “Advanced Java bytecode instrumentation”. In: *Proceedings of the 5th international symposium on Principles and practice of programming in Java*. ACM. 2007, pp. 135–144.

[12] Rex Black, Erik van Veenendaal, and Dorothy Graham. *Foundations of Software Testing: ISTQB Certification.* Cengage Learning, 2012. ISBN: 9781408044056.

[13] Peggy Cellier, Mireille Ducassé, Sébastien Ferré, and Olivier Ridoux. "Formal concept analysis enhances fault localization in software". In: *Lecture Notes in Computer Science* 4933 (2008), pp. 273–288.

[14] *Cobertura.* `http://cobertura.github.io/cobertura/`. (Accessed on 2023-01-02).

[15] J. S. Collofello and L. Cousins. "Towards automatic software fault location through decision-to-decision path analysis". In: *Managing Requirements Knowledge, International Workshop on(AFIPS).* Vol. 00. Dec. 1899, p. 539.

[16] W. J. Conover. *Practical Nonparametric Statistics.* 3rd Edition. Wiley, 1998.

[17] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. "Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact". English. In: *Empirical Software Engineering* 10.4 (2005), pp. 405–435. ISSN: 1382-3256.

[18] Pär Emanuelsson and Ulf Nilsson. "A Comparative Study of Industrial Static Analysis Tools". In: *Electron. Notes Theor. Comput. Sci.* 217 (July 2008), pp. 5–21.

[19] Francesca Arcelli Fontana, Elia Mariani, Andrea Mornioli, Raul Sormani, and Alberto Tonello. "An Experience Report on Using Code Smells Detection Tools". In: *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops.* ICSTW '11. IEEE Computer Society, 2011, pp. 450–457.

[20] Richard Herbert Franke and James D. Kaul. "The Hawthorne Experiments: First Statistical Interpretation". In: *American Sociological Review* 43.5 (1978), pp. 623–643. ISSN: 00031224. URL: `http://www.jstor.org/stable/2094540`.

[21] Gordon Fraser and Andrea Arcuri. "EvoSuite: Automatic Test Suite Generation for Object-oriented Software". In: *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering.* ESEC/FSE '11. Szeged, Hungary: ACM, 2011, pp. 416–419. ISBN: 978-1-4503-0443-6.

[22] Peter Fritzson, Nahid Shahmehri, Mariam Kamkar, and Tibor Gyimothy. "Generalized algorithmic debugging and testing". In: *ACM Letters on Programming Languages and Systems* 1.4 (Dec. 1992), pp. 303–322. ISSN: 10574514.

[23] Z. P. Fry and W. Weimer. "A human study of fault localization accuracy". In: *2010 IEEE International Conference on Software Maintenance.* Sept. 2010, pp. 1–10. DOI: `10.1109/ICSM.2010.5609691`.

[24] *gcov-—a Test Coverage Program.* `https://gcc.gnu.org/onlinedocs/gcc/Gcov.html`. (Accessed on 2023-01-02).

[25] L. Gong, D. Lo, L. Jiang, and H. Zhang. "Interactive fault localization leveraging simple user feedback". In: *2012 28th IEEE International Conference on Software Maintenance (ICSM).* Sept. 2012, pp. 67–76.

[26] Liang Gong, David Lo, Lingxiao Jiang, and Hongyu Zhang. "Interactive fault localization leveraging simple user feedback". In: *IEEE International Conference on Software Maintenance, ICSM.* IEEE, 2012, pp. 67–76. ISBN: 9781467323123.

[27] Alberto Gonzalez-Sanchez, Rui Abreu, Hans-Gerhard Gross, and Arjan J. C. van Gemund. “Prioritizing tests for fault localization through ambiguity group reduction.” In: *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on.* Ed. by Perry Alexander, Corina S. Pasareanu, and John G. Hosking. IEEE, Nov. 2011, pp. 83–92.

[28] Todd L. Graves, Mary Jean Harrold, Jung-Min Kim, Adam Porter, and Gregg Rothermel. “An Empirical Study of Regression Test Selection Techniques”. In: *ACM Trans. Softw. Eng. Methodol.* 10.2 (Apr. 2001), pp. 184–208. ISSN: 1049-331X.

[29] Robert J. Grissom and John J. Kim. *Effect sizes for research: A broad practical approach.* 2nd Edition. Lawrence Earlbaum Associates, 2005.

[30] Péter Gyimesi, Béla Vancsics, Andrea Stocco, Davood Mazinanian, Árpád Beszédes, Rudolf Ferenc, and Ali Mesbah. “BugJS: A Benchmark of JavaScript Bugs”. In: *Proceedings of 12th IEEE International Conference on Software Testing, Verification and Validation (ICST).* 2019.

[31] Dan Hao, Lu Zhang, Tao Xie, Hong Mei, and Jia-Su Sun. “Interactive Fault Localization Using Test Information”. In: *Journal of Computer Science and Technology* 24.5 (Sept. 2009), pp. 962–974. ISSN: 1000-9000.

[32] Mary Jean Harrold, Gregg Rothermel, Kent Sayre, Rui Wu, and Liu Yi. “An empirical investigation of the relationship between spectra differences and regression faults”. In: *Software Testing, Verification and Reliability* 10.3 (2000), pp. 171–194.

[33] Mary Jean Harrold, Gregg Rothermel, Rui Wu, and Liu Yi. “An empirical investigation of program spectra”. In: *Proc. of the 1998 ACM SIGPLAN-SIGSOFT workshop PASTE ’98.* Montreal, Quebec, Canada: ACM, 1998, pp. 83–90. ISBN: 1-58113-055-4.

[34] S. Holm. “A Simple Sequentially Rejective Bonferroni Test Procedure”. In: *Scandinavian Journal on Statistics* 6 (1979), pp. 65–70.

[35] *iFL 4 Eclipse.* `https://github.com/InteractiveFaultLocalization/iFL4Eclipse`. (Accessed on 2023-01-02).

[36] Laura Inozemtseva and Reid Holmes. “Coverage is Not Strongly Correlated with Test Suite Effectiveness”. In: *Proceedings of the International Conference on Software Engineering.* 2014.

[37] *JavaParser - homepage.* `https://javaparser.org/`. (Accessed on 2023-01-02).

[38] *Javassist.* `http://jboss-javassist.github.io/javassist/`. (Accessed on 2023-01-02).

[39] Yue Jia and Mark Harman. “An Analysis and Survey of the Development of Mutation Testing”. In: *Software Engineering, IEEE Transactions on* 37.5 (Sept. 2011), pp. 649–678.

[40] James A. Jones and Mary Jean Harrold. “Empirical evaluation of the tarantula automatic fault-localization technique”. In: *Proc. of International Conference on Automated Software Engineering.* Long Beach, CA, USA: ACM, 2005, pp. 273–282. ISBN: 1-58113-993-4.

[41] René Just, Darioush Jalali, and Michael D Ernst. "Defects4J: A database of existing faults to enable controlled testing studies for Java programs". In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis.* ACM. 2014, pp. 437–440.

[42] René Just, Darioush Jalali, Laura Inozemtseva, Michael D Ernst, Reid Holmes, and Gordon Fraser. "Are mutants a valid substitute for real faults in software testing?" In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering.* ACM. 2014, pp. 654–665.

[43] Elinda Kajo-Mece and Megi Tartari. "An Evaluation of Java Code Coverage Testing Tools". In: *Proceedings of the 2012 Balkan Conference in Informatics (BCI'12).* Faculty of Sciences, University of Novi Sad, 2012, pp. 72–75. ISBN: ISBN 978-86-7031-200-5.

[44] Mehdi Kessis, Yves Ledru, and Gérard Vandome. "Experiences in Coverage Testing of a Java Middleware". In: *Proceedings of the 5th International Workshop on Software Engineering and Middleware.* SEM '05. Lisbon, Portugal: ACM, 2005, pp. 39–45. ISBN: 1-59593-205-4.

[45] Ákos Kiss, Renáta Hodován, and Tibor Gyimóthy. "Coarse Hierarchical Delta Debugging". In: *Proceedings of the 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME).* Sept. 2017, pp. 194–203.

[46] Andrew Jensen Ko and Brad A. Myers. "Designing the whyline: a debugging interface for asking questions about program behavior". In: *Proceedings of the 2004 Conference on Human Factors in Computing Systems, CHI 2004, Vienna, Austria, April 24 - 29, 2004.* 2004, pp. 151–158.

[47] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. "Practitioners' expectations on automated fault localization". In: *Proceedings of the 25th International Symposium on Software Testing and Analysis - ISSTA 2016.* New York, New York, USA: ACM Press, 2016, pp. 165–176. ISBN: 9781450343909.

[48] Tien-Duy B. Le, Ferdian Thung, and David Lo. "Theory and Practice, Do They Match? A Case with Spectrum-Based Fault Localization". In: *2013 IEEE International Conference on Software Maintenance.* Sept. 2013, pp. 380–383. ISBN: 978-0-7695-4981-1.

[49] Xuan-Bach D. Le, Lingfeng Bao, David Lo, Xin Xia, Shanping Li, and Corina Pasareanu. "On Reliability of Patch Correctness Assessment". In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE).* 2019, pp. 524–535. DOI: 10.1109/ICSE.2019.00064.

[50] Daniel Lehmann and Michael Pradel. "Feedback-directed Differential Testing of Interactive Debuggers". In: *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* ESEC/FSE 2018. Lake Buena Vista, FL, USA: ACM, 2018, pp. 610–620. ISBN: 978-1-4503-5573-5.

[51] Yan Lei, Xiaoguang Mao, Ziying Dai, and Dengping Wei. "Effective Fault Localization Approach Using Feedback". In: *IEICE Transactions on Information and Systems* E95.D.9 (2012), pp. 2247–2257.

[52] Hang LI. “A Short Introduction to Learning to Rank”. In: *IEICE Transactions on Information and Systems* E94.D.10 (2011), pp. 1854–1862.

[53] Nan Li, Xin Meng, J. Offutt, and Lin Deng. “Is bytecode instrumentation as good as source code instrumentation: An empirical study with industrial tools (Experience Report)”. In: *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on.* Nov. 2013, pp. 380–389.

[54] Xiangyu Li, Marcelo d’Amorim, and Alessandro Orso. “Iterative User-Driven Fault Localization”. In: *Hardware and Software: Verification and Testing: 12th International Haifa Verification Conference, HVC 2016, Haifa, Israel, November 14-17, 2016, Proceedings.* Cham: Springer International Publishing, 2016, pp. 82–98. ISBN: 978-3-319-49052-6.

[55] Xiangyu Li, Shaowei Zhu, Marcelo d’Amorim, and Alessandro Orso. “Enlightened Debugging”. In: *Proceedings of the 40th IEEE and ACM SIGSOFT International Conference on Software Engineering (ICSE 2018).* Gothenburg, Sweden: ACM, 2018.

[56] Ben Liblit, Mayur Naik, Alice X Zheng, Alex Aiken, and Michael I Jordan. “Scalable statistical bug isolation”. In: *Acm Sigplan Notices* 40.6 (2005), pp. 15–26.

[57] Yun Lin, Jun Sun, Yinxing Xue, Yang Liu, and Jinsong Dong. “Feedback-based Debugging”. In: *Proceedings of the 39th International Conference on Software Engineering.* Buenos Aires, Argentina: IEEE Press, 2017, pp. 393–403. ISBN: 978-1-5386-3868-2.

[58] Raghu Lingampally, Atul Gupta, and Pankaj Jalote. “A multipurpose code coverage tool for Java”. In: *System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International Conference on.* IEEE. 2007, 261b–261b.

[59] Chao Liu, Long Fei, Xifeng Yan, Jiawei Han, and Samuel P Midkiff. “Statistical debugging: A hypothesis testing-based approach”. In: *IEEE Transactions on software engineering* 32.10 (2006), pp. 831–848.

[60] Wes Masri, Rawad Abou-Assi, Marwa El-Ghali, and Nour Al-Fatairi. “An Empirical Study of the Factors That Reduce the Effectiveness of Coverage-based Fault Localization”. In: *Proceedings of the 2nd International Workshop on Defects in Large Software Systems.* DEFECTS ’09. Chicago, Illinois: ACM, 2009, pp. 1–5. ISBN: 978-1-60558-654-0.

[61] Wes Masri and Rawad Abou Assi. “Prevalence of Coincidental Correctness and Mitigation of Its Impact on Fault Localization”. In: *ACM Trans. Softw. Eng. Methodol.* 23.1 (Feb. 2014), 8:1–8:28. ISSN: 1049-331X.

[62] Wolfgang Mayer and Markus Stumptner. “Evaluating models for model-based debugging”. In: *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering.* IEEE Computer Society. 2008, pp. 128–137.

[63] S. C. Ntafos. “A comparison of some structural testing strategies”. In: *IEEE Transactions on Software Engineering* 14.6 (June 1988), pp. 868–874.

[64] A. Jefferson Offutt, Jie Pan, and Jeffrey M. Voas. “Procedures for reducing the size of coverage-based test sets”. In: *In Proc. Twelfth Int’l. Conf. Testing Computer Softw.* 1995, pp. 111–123.

[65] Alessandro Orso, James A. Jones, Mary Jean Harrold, and John T. Stasko. "Gammatella: Visualization of Program-Execution Data for Deployed Software". In: *26th International Conference on Software Engineering (ICSE 2004), 23-28 May 2004, Edinburgh, United Kingdom*. 2004, pp. 699–700.

[66] Thomas Ostrand. "White-Box Testing". In: *Encyclopedia of Software Engineering* (2002).

[67] Mike Papadakis and Yves Le Traon. "Effective fault localization via mutation analysis". In: *Proceedings of the 29th Annual ACM Symposium on Applied Computing - SAC '14*. New York, New York, USA: ACM Press, 2014, pp. 1293–1300. ISBN: 9781450324694.

[68] Mike Papadakis and Yves Le Traon. "Metallaxis-FL: mutation-based fault localization". In: *Software Testing, Verification and Reliability* 25.5-7 (Aug. 2015), pp. 605–628. ISSN: 09600833.

[69] Priya Parmar and Miral Patel. "Software Fault Localization: A Survey". In: *Intl. Journal of Computer Applications* 154.9 (2016), pp. 6–13.

[70] Chris Parnin and Alessandro Orso. "Are Automated Debugging Techniques Actually Helping Programmers?" In: *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. Toronto, Ontario, Canada: ACM, 2011, pp. 199–209. ISBN: 978-1-4503-0562-4.

[71] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, and Benjamin Keller. "Evaluating and improving fault localization". In: *Proceedings of the 39th International Conference on Software Engineering* (2017), pp. 609–620.

[72] Alexandre Perez and Rui Abreu. "A Diagnosis-based Approach to Software Comprehension". In: *Proceedings of the 22nd International Conference on Program Comprehension*. ICPC 2014. Hyderabad, India: ACM, 2014, pp. 37–47. ISBN: 978-1-4503-2879-1.

[73] Leandro Sales Pinto, Saurabh Sinha, and Alessandro Orso. "Understanding Myths and Realities of Test-suite Evolution". In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 2012, 33:1–33:11.

[74] Päivi Raulamo-Jurvanen. "Decision Support for Selecting Tools for Software Test Automation". In: *SIGSOFT Softw. Eng. Notes* 41.6 (Jan. 2017), pp. 1–5.

[75] S. Rayadurgam and M.P.E. Heimdahl. "Coverage based test-case generation using model checkers". In: *Engineering of Computer Based Systems, 2001. ECBS 2001. Proceedings. Eighth Annual IEEE International Conference and Workshop on the*. 2001, pp. 83–91.

[76] Manos Renieris and Steven P. Reiss. "Fault Localization With Nearest Neighbor Queries". In: *Proceedings of 18th IEEE International Conference on Automated Software Engineering (ASE 2003)*. IEEE Computer Society, 2003, pp. 30–39.

[77] Thomas Reps, Thomas Ball, Manuvir Das, and James Larus. "The Use of Program Profiling for Software Maintenance with Applications to the Year 2000 Problem". In: *ACM SIGSOFT Software Engineering Notes* 22.6 (Nov. 1997), pp. 432–449.

[78] *Research about Interactive Fault Localization.* `https://interactivefaultlocalization.github.io/`. (Accessed on 2023-01-02).

[79] Jorge Ressia, Alexandre Bergel, and Oscar Nierstrasz. "Object-centric Debugging". In: *Proceedings of the 34th International Conference on Software Engineering.* ICSE '12. Zurich, Switzerland: IEEE Press, 2012, pp. 485–495. ISBN: 978-1-4673-1067-3.

[80] Gregg Rothermel, Mary Jean Harrold, Jeffery von Ronne, and Christie Hong. "Empirical studies of test-suite reduction". In: *Software Testing, Verification and Reliability* 12.4 (2002), pp. 219–249. ISSN: 1099-1689.

[81] Gregg Rothermel, Roland J. Untch, and Chengyun Chu. "Prioritizing Test Cases For Regression Testing". In: *IEEE Trans. Softw. Eng.* 27.10 (Oct. 2001), pp. 929–948. ISSN: 0098-5589.

[82] Atanas Rountev, Scott Kagan, and Michael Gibas. "Static and Dynamic Analysis of Call Chains in Java". In: *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis.* ACM. New York, NY, USA: ACM, 2004. ISBN: 1-58113-820-2. DOI: `10.1145/1007512.1007514`. URL: `http://doi.acm.org/10.1145/1007512.1007514`.

[83] Ripon K. Saha, Yingjun Lyu, Wing Lam, Hiroaki Yoshida, and Mukul R. Prasad. "Bugs.Jar: A Large-scale, Diverse Dataset of Real-world Java Bugs". In: *Proceedings of the 15th International Conference on Mining Software Repositories.* MSR '18. Gothenburg, Sweden: ACM, 2018, pp. 10–13. ISBN: 978-1-4503-5716-6.

[84] A. Schröter, N. Bettenburg, and R. Premraj. "Do stack traces help developers fix bugs?" In: *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010).* May 2010, pp. 118–121.

[85] Sina Shamshiri, Rene Just, Jose Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. "Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (t)". In: *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on.* IEEE. 2015, pp. 201–211.

[86] Josep Silva. "A Survey on Algorithmic Debugging Strategies". In: *Adv. Eng. Softw.* 42.11 (Nov. 2011), pp. 976–991. ISSN: 0965-9978.

[87] *SoDA library.* `http://soda.sed.hu`. (Accessed on 2023-01-02).

[88] Jeongju Sohn and Shin Yoo. "FLUCCS: Using Code and Change Metrics to Improve Fault Localization". In: *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis.* ISSTA 2017. ACM, 2017, pp. 273–283.

[89] Higor Amario de Souza, Marcos Lordello Chaim, and Fabio Kon. "Spectrum-based Software Fault Localization: A Survey of Techniques, Advances, and Challenges". In: *CoRR* abs/1607.04347 (2016). arXiv: `1607.04347`.

[90] Friedrich Steimann, Marcus Frenkel, and Rui Abreu. "Threats to the Validity and Value of Empirical Assessments of the Accuracy of Coverage-based Fault Locators". In: *Proceedings of the 2013 International Symposium on Software Testing and Analysis.* Lugano, Switzerland: ACM, 2013, pp. 314–324. ISBN: 978-1-4503-2159-4.

[91] Dávid Tengeri, Árpád Beszédes, Dávid Havas, and Tibor Gyimóthy. “Toolset and Program Repository for Code Coverage-Based Test Suite Analysis and Manipulation”. In: *Proceedings of the 14th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM’14)*. Victoria, City of Gardens, British Columbia, Canada, Sept. 2014, pp. 47–52.

[92] Macario Polo Usaola and Pedro Reales Mateo. “Mutation Testing Cost Reduction Techniques: A Survey”. In: *IEEE Software* 27.3 (2010), pp. 80–86. ISSN: 0740-7459.

[93] L. Vidács, Á. Beszédes, D. Tengeri, I. Siket, and T. Gyimóthy. “Test suite reduction for fault detection and localization: A combined approach”. In: *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week - IEEE Conference on.* Feb. 2014, pp. 204–213.

[94] Jeffrey M. Voas. “PIE: A Dynamic Failure-Based Technique”. In: *IEEE Trans. Softw. Eng.* 18.8 (Aug. 1992), pp. 717–727. ISSN: 0098-5589.

[95] Xinming Wang, S. C. Cheung, W. K. Chan, and Zhenyu Zhang. “Taming Coincidental Correctness: Coverage Refinement with Context Patterns to Improve Fault Localization”. In: *Proceedings of the 31st International Conference on Software Engineering.* ICSE ’09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 45–55. ISBN: 978-1-4244-3453-4.

[96] W Eric Wong and Vidroha Debroy. “A survey of software fault localization”. In: *Department of Computer Science, University of Texas at Dallas, Tech. Rep. UTDCS-45* 9 (2009).

[97] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. “A survey on software fault localization”. In: *IEEE Transactions on Software Engineering* 42.8 (2016), pp. 707–740.

[98] W Eric Wong and Yu Qi. “BP neural network-based effective fault localization”. In: *International Journal of Software Engineering and Knowledge Engineering* 19.04 (2009), pp. 573–597.

[99] W. Eric Wong, Vidroha Debroy, Ruizhi Gao, and Yihao Li. “The DStar Method for Effective Software Fault Localization”. In: *IEEE Trans. Reliability* 63 (2014), pp. 290–308.

[100] Franz Wotawa, Markus Stumptner, and Wolfgang Mayer. “Model-based debugging or how to diagnose programs automatically”. In: *IEA/AIE.* Springer. 2002, pp. 746–757.

[101] Rongxin Wu, Hongyu Zhang, Shing-Chi Cheung, and Sunghun Kim. “CrashLocator: Locating Crashing Faults Based on Crash Stacks”. In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis.* ISSTA 2014. San Jose, CA, USA: ACM, 2014, pp. 204–214. ISBN: 978-1-4503-2645-2.

[102] Xin Xia, Lingfeng Bao, David Lo, and Shanping Li. ““Automated Debugging Considered Harmful” Considered Harmful: A User Study Revisiting the Usefulness of Spectra-Based Fault Localization Techniques with Professionals Using Real Bugs from Large Systems”. In: *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME).* IEEE, Oct. 2016, pp. 267–278. ISBN: 978-1-5090-3806-0.

[103] Xiaoyuan Xie, Tsong Yueh Chen, Fei-Ching Kuo, and Baowen Xu. “A Theoretical Analysis of the Risk Evaluation Formulas for Spectrum-Based Fault Localization”. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* (2013). In press.

[104] Xiaofeng Xu, Vidroha Debroy, W. Eric Wong, and Donghui Guo. “Ties within Fault Localization rankings: Exposing and Addressing the Problem”. In: *International Journal of Software Engineering and Knowledge Engineering* 21 (2011), pp. 803–827.

[105] J. Xuan and M. Monperrus. “Learning to Combine Multiple Ranking Metrics for Fault Localization”. In: *2014 IEEE International Conference on Software Maintenance and Evolution*. Sept. 2014, pp. 191–200.

[106] Qian Yang, J Jenny Li, and David M Weiss. “A survey of coverage-based testing tools”. In: *The Computer Journal* 52.5 (2009), pp. 589–597.

[107] S. Yoo and M. Harman. “Regression testing minimization, selection and prioritization: a survey”. In: *Software Testing, Verification and Reliability* 22.2 (2012), pp. 67–120. ISSN: 1099-1689.

[108] Shin Yoo. “Evolving human competitive spectra-based fault localisation techniques”. In: *Proceedings of the 4th international conference on Search Based Software Engineering*. SSBSE’12. Riva del Garda, Italy: Springer-Verlag, 2012, pp. 244–258. ISBN: 978-3-642-33118-3.

[109] Shin Yoo, Mark Harman, and David Clark. “Fault localization prioritization: Comparing information-theoretic and coverage-based approaches”. In: *ACM Trans. Softw. Eng. Methodol.* 22.3 (2013), p. 19.

[110] Shin Yoo, Mark Harman, and David Clark. *FLINT: Fault Localisation using Information Theory*. Tech. rep. University College London, 2011.

[111] Shin Yoo, Xiaoyuan Xie, Fei-Ching Kuo, Tsong Yueh Chen, and Mark Harman. “Human Competitiveness of Genetic Programming in Spectrum-Based Fault Localisation: Theoretical and Empirical Analysis”. In: *ACM Trans. Softw. Eng. Methodol.* 26.1 (June 2017), 4:1–4:30. ISSN: 1049-331X.

[112] Andreas Zeller. *Why Programs Fail, Second Edition: A Guide to Systematic Debugging*. 2nd. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2009, p. 424. ISBN: 9780123745156.

[113] Andreas Zeller and Ralf Hildebrandt. “Simplifying and Isolating Failure-Inducing Input”. In: *IEEE Trans. Softw. Eng.* 28.2 (Feb. 2002), pp. 183–200. ISSN: 0098-5589.

[114] Mengshi Zhang, Xia Li, Lingming Zhang, and Sarfraz Khurshid. “Boosting Spectrum-based Fault Localization Using PageRank”. In: *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2017. Santa Barbara, CA, USA: ACM, 2017, pp. 261–272. ISBN: 978-1-4503-5076-1.

[115] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. “Locating faults through automated predicate switching”. In: *Proceedings of the 28th international conference on Software engineering*. ACM. 2006, pp. 272–281.

[116] Xiangyu Zhang, Haifeng He, Neelam Gupta, and Rajiv Gupta. "Experimental evaluation of using dynamic slices for fault location". In: *Proceedings of the sixth international symposium on Automated analysis-driven debugging*. ACM. 2005, pp. 33–42.

[117] Daming Zou, Jingjing Liang, Yingfei Xiong, Michael D. Ernst, and Lu Zhang. *An Empirical Study of Fault Localization Families and Their Combinations*. arXiv:1803.09939 [cs.SE]. Feb. 2018.