Code Coverage Measurement and Fault Localization Approaches

THESES OF THE PH.D. THESIS

by

Ferenc Horváth

Supervisor:

Árpád Beszédes, Ph.D.
associate professor

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

Szeged, 2023

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.

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.

Code Coverage Measurement

Thesis I – 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 of the dissertation.

Software testers have long established the theory and practice of code coverage measurement: various types of coverage criteria like statement, branch, and others [4], as well as technical solutions including various kinds of instrumentation methods [24]. 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 [24]. 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

Fault Localization

Thesis II – Interactive Fault Localization

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

Recent studies highlighted some barriers to the wide adoption of the SBFL methods, including a high number of suggested elements to investigate [22, 14], applicability of theoretical results in practice [11], little experimental results with real faults [15], validity issues of empirical research [18], 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 [15, 22, 14, 10]. 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 [20, 15, 23, 25]. 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 [8], Ochiai [1], and DStar [21]. We relied on two benchmarks: artificial defects from the SIR repository [5] and real defects from Defects4J [9]. 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. [6], 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

Thesis III – Call-Chain-Based Fault Localization

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

The state-of-the-art approach to SBFL is to use the so-called "hit-based" spectra [7] 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 [20, 15, 23]. Popular suspiciousness scores are Tarantula [8], Ochiai [1], and DStar [21], among others.

One reason why an SBFL formula may fail is what is referred to as *coincidental correctness* [19, 12, 3]. 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 [13], and if there are relatively more such cases than traversing and failing ones, the suspiciousness score will be negatively affected [12].

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 [26], or by involving additional information to the process. Early attempts to incorporate control or data flow information, for instance [16, 7], 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. [17], and Zou et al. [26] 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 Beszédes et al. [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 [1].

Our approach works at a higher granularity than statement-level approaches (previous work suggests that function-level is a suitable granularity for the users [2, 26]). 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 [9]. 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

Contributions of the Theses

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 the aforementioned challenges.

Thesis I is discussed in detail in **Chapter 3**, and it responds to **challenges C1** and **C2** discussed above. It considers 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.

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.

Thesis II is discussed in detail in **Chapter 4**, and it responds to **challenges C3** and **C4** discussed above. It 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.

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.

Thesis III is discussed in detail in **Chapter 5**, and it responds to **challenge C3** discussed above. It 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.

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.

Table 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 1: *Thesis contributions and supporting publications*

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, 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.

[2] 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.

[3] 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.

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

Összefoglalás

A kódlefedettség mérés fontos szerepet játszik az ipari gyakorlatban és a tudományos kutatásokban is. Számos terület függ a kódlefedettségtől, például a teszt esetek generálása, a tesztek priorizálása és a hibalokalizáció is. Ezek közül az értekezés két fő témára fókuszál, és a tézispontok ennek megfelelően két részre oszlanak.

Az értekezés első részének 3. fejezete a Java kódlefedettség mérésére szolgáló eszközök (forráskód- és bájtkód alapú) által adott lefedettségi eredmények összehasonlításáról számol be. A kísérleteket 5 népszerű eszközzel, 8 nyílt forráskódú programon és metódus szinten végeztük. Megállapítottuk, hogy a lefedettségibeli különbségek 1,5% és 40% közöttiek és mindkét irányban fellépnek. A különbségeket részletesen is vizsgáltunk az egyes teszt esetek és programelemek szemszögéből. A kódlefedettség számos alkalmazásánál (például a hibakeresésnél) az ilyen szintű finom különbségek komoly zavart okozhatnak. Az egyes tesztesetek között akár 14%-os, a metódusok között pedig több mint 20%-os különbségeket mértünk. Szisztematikusan megvizsgáltuk a különbségek okait, és azt találtuk, hogy a különbségek egy része eszköz-specifikus, míg a többi a mérési megközelítésnek tulajdonítható. Az 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 és azt találtuk, hogy a priorizált listák jelentősen eltértek egymástól. Az alacsony korreláció kockázatot jelent, mert megjósolhatatlan egy adott lefedettségi pontatlanság potenciális felerősödése egy adott alkalmazásban.

Az automatikus hibakeresés eszköztámogatása korlátozott, mivel gyakran a legkorszerűbb algoritmusok sem nyújtanak hatékony segítséget a felhasználónak. Általában csupán a gyanús kódelemek rangsorolt listáját kínálják és a hiba jellemzően nem a rangsor elején található. A spektrumalapú hibalokalizáció (a kódlefedettséget és a tesztek eredményeit használja a rangsor kiszámításához) esetén a fejlesztőnek jellemzően több elemet is meg kell vizsgálnia, mielőtt megtalálja a valóban hibásat. Az értekezés második részének 4. fejezete egy olyan megközelítést javasol, amelyben a fejlesztő interakcióba léphet az algoritmussal azáltal, hogy visszajelzést ad a rangsorolt lista elemeiről. Ezzel kihasználjuk a felhasználó tudását a rangsorolt lista aktuális elemének környezetéről, így akár nagyobb kódegységek is átpozícionálhatóak. A megközelítést először szimulált felhasználókkal értékeltük ki, ahol kétféle “bizonytalanságot” modelleztünk: a tudást és magabiztosságot. Az eredmények még erős felhasználói bizonytalanság esetén is figyelemre méltó javulást mutattak a hibalokalizáció hatékonyságában. Ezután a megközelítés hatékonyságát valódi felhasználókkal is vizsgáltuk és az eredmények alapján ígéretesnek bizonyult a módszer.

Spektrumalapú hibalokalizáció során a programelemeket egy gyanússági érték alapján rangsorolják, amely bár segítheti a programozót a hiba megtalálásában, de nem ad semmilyen információt a gyanús kódelemekről. Bár történtek kísérletek arra, hogy a folyamatba vezérlési- vagy adatfolyam alapú információkat építsenek be, ezek a skálázhatósági problémák miatt nem jártak sikerrel. Az értekezés második részének 5. fejezete a spektrumalapú hibalokalizáció kiegészítését javasolja függvényhívási láncokkal (azaz a hívási verem pillanatfelvételeivel), amelyeken először elvégezzük a hibalokalizációt, majd az eredményeket visszavezetjük a függvényekre. A valós programokkal végzett kísérletek azt mutatják, hogy a javasolt módszer hatékonysága jelentősen jobb a hagyományos függvényszintű megközelítéshez képest és a számítási többletköltség is kezelhető mértékű.

Declaration

In the Ph.D. dissertation of Ferenc Horváth entitled "*Code Coverage Measurement and Fault Localization Approaches*", Ferenc Horváth and the corresponding co-authors share the following joint and undividable contributions:

- Thesis I: overview of theoretical differences in code coverage measurement tools for Java; collection, categorization, testing, and selection of code coverage measurement tools; collection, configuration, and selection of Java programs on which the experiments were executed; measurement and analysis of the differences in code coverage of Java bytecode and source code instrumentation tools; systematic investigation of discrepancies in coverage data and their causes, and development of fixes and recommendations for the correction of the issues [c3, j2].

- Thesis II: development of the theoretical background of applying the concept of interactive feedback in FL; development of the theoretical background of applying the concept of user imperfection factors (confidence and knowledge) in the evaluation of FL; measurement and analysis of the effectiveness and efficiency of *iFL* in the simulated environment; reimplementation of the TALK algorithm, and the execution of the subsequent comparative experiments and analyses; design and development of the iFL4Eclipse plug-in which implements *iFL* in the Eclipse IDE; design, execution and analysis of the user studies [j1, c2, w1, t1, p1]

- Thesis III: development of the theoretical background of applying the concept of function call chains in SBFL; measurment and analysis of the effectiveness and efficiency of this method [c1, c4, j3].

In the Ph.D. dissertation of Ferenc Horváth entitled "*Code Coverage Measurement and Fault Localization Approaches*", Ferenc Horváth's contribution was decisive in the following results:

- Thesis I: design, implementation, and execution of test prioritization and selection experiments; organization, pre-processing, analysis, evaluation, and presentation of measurement data on test prioritization and selection [c3, j2].

- Thesis II: design and implemention of the simulation framework as a basis for testing interactive fault localization approaches; implementation of the *iFL* approach in the simulation framework; execution of experiments using seeded and real faults from the SIR and Defects4J benchmark to evaluate *iFL* [j1, c2, w1, t1, p1].

- Thesis III: design and implementation of a bytecode instrumentation tool that was used to collect the call chains in the experiments; implementation of the call-chain-based FL approach, including the weighted chain count, the reapplied spectrum, and the rank merging algorithms; execution of experiments using real faults from the Defects4J benchmark to evaluate this approach [c1, c4, j3].

These results cannot be used to obtain an academic research degree, other than the submitted Ph.D. thesis of Ferenc Horváth.

Szeged, 2023.01.24.

Ferenc Horváth	Árpád Beszédes, Ph.D.
Ph.D. Candidate	*Supervisor*

The head of the Doctoral School of Computer Science declares that the declaration above was sent to all of the co-authors and none of them raised any objections against it.

Szeged, 2023. január 25.

SZEGEDI TUDOMÁNYEGYETEM INFORMATIKAI DOKTORI ISKOLA SZEGED

Márk Jelasity, D.Sc.
Head of Doctoral School