Assessment of Spectrum-based Fault Localization for Practical Use

Higor Amario de Souza

THESIS PRESENTED TO THE INSTITUTE OF MATHEMATICS AND STATISTICS OF THE UNIVERSITY OF SÃO PAULO TO THE TITLE OF DOCTOR IN COMPUTER SCIENCE

Graduate Program in Computer Science

Advisor: Prof. Fabio Kon

This research received financial support from the São Paulo Research Foundation (FAPESP), processes 2013/24992-2 and 2014/23030-5, and from the Coordination for the Improvement of Higher Education (CAPES).

São Paulo, April 2018
Assessment of Spectrum-based Fault Localization for Practical Use

This version of the thesis contains corrections and changes suggested by the Judging Committee during the defense of the original version of the work, held on April 17, 2018. A copy of the original version is available at the Institute of Mathematics and Statistics of the University of São Paulo.

Doctoral Thesis Committee:

- Prof. Dr. Adenilso da Silva Simão - ICMC-USP
- Prof. Dr. Alfredo Goldman Vel Lejbman - IME-USP
- Prof. Dr. Auri Marcelo Rizzo Vincenzi - UFSCar
- Prof. Dr. Marcos Lordello Chaim - EACH-USP
- Prof. Dr. Otávio Augusto Lazzarini Lemos - Unifesp
Acknowledgements

I would like to express my gratitude and appreciation to the persons below who assisted and supported me in various ways during this research:

My wife Rose and my son Gustavo for loving and supporting me at every moment. Thank you for understanding my moments of absence. I owe everything to you!

My parents Regina and Osvaldo, and my brother Raphael, whose love, support, encouragement, and example are central to my life.

My advisor, Professor Fabio Kon, whose support and guidance allowed me to carry out this research. You are a great advisor. Thank you for helping me to focus on the things that really matter and that I must be aware of.

My co-advisor, Professor Marcos Lordello Chaim for his guidance, insights, and encouragement. You are a fundamental part of this work. It has been a long time working together since the Master’s Degree. Thanks for his friendship, help, motivation, and support through these years.

Professor Marcelo Lauretto for his important collaboration in the design of our user study.

Professors Adenilso Simão, Alfredo Goldman, Auri Vincenzi, and Otávio Lemos for their review, comments, suggestions, and insights that helped me to improve the final version of this thesis.

Professors Marcelo D’Amorim and Marco Aurélio Gerosa for their valuable feedback in my qualifying exam, which helped me to improve this work.

Professors Rafael Oliveira, Ivanilton Polato, Eduardo Guerra, Raphael Camargo, Márcio Delamaro, and Carlos Eduardo Ferreira, who helped me to disclose our user study for their students. Also, thanks for all the participants of the user study for their time and feedback.

Professors Daniel Batista and Kelly Braghetto for their feedback and suggestions during our group meetings.

My colleagues from IME-USP and EACH-USP, namely, Roberto Andrioli, Henrique Ribeiro, Danilo Mutti, Nelson Lago, Carlos Santos, Paulo Moura, Marcos González, Monna Santos, Maurício Aniche, Daniel Cukier, Arthur Esposte, Athos Ribeiro, Felipe Guimarães, Eduardo Santana, Diego Camarinha, Elaine Watanabe, Graziela Tonin, Albert Fuente, Renan Oliveira, Luiz Batistão, and Fabio Silva. Every one of you helped me in various ways, with advice, feedback, insights, support, and laughs through these Ph.D. years.

Professor Alessandro Orso for kindly accepting me as a visiting student at Georgia Tech. That was a great experience in my life. Also, I thank my labmates Qianqian Wang, Xiangyu Li, Mattia Fazzini, Shauvik Roy Choudhary, Eduardo Noronha, Farnaz Behrang, and Xin Zhang for their support during my stay in Atlanta.

Finally, FAPESP and CAPES for the financial support, which was fundamental for this research.
Abstract


Debugging is one of the most time-consuming activities in software development. Several fault localization techniques have been proposed in the last years, aiming to reduce development costs. A promising approach, called Spectrum-based Fault localization (SFL), consists of techniques that provide a list of suspicious program elements (e.g., statements, basic blocks, methods) more likely to be faulty. Developers should inspect a suspiciousness list to search for faults. However, these fault localization techniques are not yet used in practice.

These techniques are based on assumptions about the developer’s behavior when inspecting such lists that may not hold in practice. A developer is supposed to inspect an SFL list from the most to the least suspicious program elements (e.g., statements) until reaching the faulty one. This assumption leads to some implications: the techniques are assessed only by the position of a bug in a list; a bug is deemed as found when the faulty element is reached.

SFL techniques should pinpoint the faulty program elements among the first picks to be useful in practice. Most techniques use ranking metrics to assign suspiciousness values to program elements executed by the tests. These ranking metrics have presented similar modest results, which indicates the need for different strategies to improve the effectiveness of SFL. Moreover, most techniques use only control-flow spectra due to the high execution costs associated with other spectra, such as data-flow.

Also, little research has investigated the use of SFL techniques by practitioners. Understanding how developers use SFL may help to clarify the theoretical assumptions about their behavior, which in turn can collaborate with the proposal of techniques more feasible for practical use. Therefore, user studies are a valuable tool for the development of the area.

The goal of this thesis research was to propose strategies to improve spectrum-based fault localization, focusing on its practical use. This thesis presents the following contributions. First, we investigate strategies to provide contextual information for SFL. These strategies helped to reduce the amount of code to be inspected until reaching the faults. Second, we carried out a user study to understand how developers use SFL in practice. The results show that developers can benefit from SFL to locate bugs. Third, we explore the use of data-flow spectrum for SFL. Data-flow spectrum singles out faults significantly better than control-flow spectrum, improving the fault localization effectiveness.

**Keywords:** Fault localization, spectrum-based, debugging, testing, contextual information, user study.
Resumo


Depuração é uma das atividades mais custosas durante o desenvolvimento de programas. Diversas técnicas de localização de defeitos têm sido propostas nos últimos anos com o objetivo de reduzir custos de desenvolvimento. Uma abordagem promissora, chamada Localização de Defeitos baseada em Espectro (LDE), é formada por técnicas que fornecem listas contendo elementos de código (comandos, blocos básicos, métodos) mais suspeitos de conter defeitos. Desenvolvedores deveriam inspecionar uma lista de suspeição para procurar por defeitos. No entanto, essas técnicas de localização de defeitos ainda não são usadas na prática.

Essas técnicas baseiam-se em suposições sobre o comportamento de desenvolvedores durante a inspeção de tais listas que podem não ocorrer na prática. Um desenvolvedor supostamente inspeciona uma lista de LDE a partir do elemento mais suspeito para o menos suspeito até atingir o elemento defeituoso. Essa suposição leva a algumas implicações: as técnicas são avaliadas somente pela posição dos defeitos nas listas; um defeito é considerado como encontrado quando o elemento defeituoso é atingido.

Técnicas de LDE deveriam posicionar os elementos de código defeituosos entre as primeiras posições para serem úteis na prática. A maioria das técnicas usa métricas de ranqueamento para atribuir valores de suspeição aos elementos executados pelos testes. Essas métricas de ranqueamento têm apresentado resultados semelhantes, o que indica a necessidade de estratégias diferentes para melhorar a eficácia de LDE. Além disso, a maioria das técnicas usa somente espectros de fluxo de controle devido ao alto custo de execução associado a outros espectros, tais como fluxo de dados.

Também, poucas pesquisas têm investigado o uso de técnicas de LDE por programadores. Entender como desenvolvedores usam LDE pode ajudar a esclarecer as suposições teóricas sobre seu comportamento, o que por sua vez pode para colaborar para a proposição de técnicas mais viáveis para uso prático. Portanto, estudos com usuários são importantes para o desenvolvimento da área.

O objetivo desta pesquisa de doutorado foi propor estratégias para melhorar a localização de defeitos baseada em espectro focando em seu uso prático. Esta tese apresenta as seguintes contribuições originais. Primeiro, nós investigamos estratégias para fornecer informação de contexto para LDE. Essas estratégias ajudaram a reduzir quantidade de código a ser inspecionado até atingir os defeitos. Segundo, nós realizamos um estudo com usuários para entender como desenvolvedores usam LDE na prática. Os resultados mostram que desenvolvedores podem beneficiar-se de LDE para localizar defeitos. Terceiro, nós exploramos o uso de espectros de fluxo de dados para LDE. Mostramos que o espectro de fluxo de dados seleciona defeitos significamente melhor que espectro.
de fluxo de controle, aumentando a eficácia de localização de defeitos.

**Palavras-chave:** Localização de defeitos, baseada em espectro, depuração, testes, informação contextual, estudo com usuários.
“... A man needs to travel. On his own, not through stories, images, books, or television. He needs to travel for himself, with his own eyes and on his own feet, to understand what is his. So that he might someday plant his own trees and value them. He must know cold in order to enjoy heat. And vice versa. He must experience distance and homelessness in order to feel at home under his own roof. A man needs to travel to places he does not know; he must lose the arrogance that causes him to see the world as he imagines it, rather than simply as it is, or as it can be; the arrogance that turns us into professors and doctors of what we have not seen, when we should be students who simply go and see.”

Amyr Klink
## Contents

List of Figures xv
List of Tables xvii

### 1 Introduction
1.1 Context and problem outline ........................................ 2
1.2 Objectives .............................................................. 3
1.3 Contributions ........................................................... 5
   1.3.1 Publications ....................................................... 6
   1.3.2 Other contributions ............................................. 7
1.4 Structure of the thesis ................................................. 7

### 2 Basic concepts
2.1 Debugging ............................................................... 9
   2.1.1 Fault, failure, error, and mistake .............................. 10
   2.1.2 Program comprehension ......................................... 10
2.2 Testing ................................................................. 10
   2.2.1 Testing strategies ................................................. 11
2.3 Debugging techniques ................................................ 14
   2.3.1 Symbolic debuggers .............................................. 14
   2.3.2 Automated debugging .......................................... 15
2.4 Spectrum-based fault localization .................................. 17
   2.4.1 Ranking metrics .................................................. 18
   2.4.2 Faults .............................................................. 19
   2.4.3 Suspiciousness lists ............................................ 20
   2.4.4 Contextual information ........................................ 20
   2.4.5 Filtering strategies ............................................. 20
   2.4.6 SFL: a running example ...................................... 21
2.5 Final remarks ......................................................... 22

### 3 Survey and related work
3.1 Survey of spectrum-based fault localization ....................... 23
   3.1.1 Selection criteria .............................................. 23
   3.1.2 Advances and challenges ..................................... 23
3.2 Related work .......................................................... 27
3.2.1 Spectrum-based fault localization ........................................... 27
3.2.2 Contextual information ....................................................... 28
3.2.3 Method-level spectrum ....................................................... 29
3.2.4 Absolute ranking ............................................................. 29
3.2.5 User studies ................................................................. 30
3.2.6 Data-flow spectrum ......................................................... 31
3.2.7 Comparison with previous work .......................................... 32

3.3 Overview of the contribution of this thesis ................................. 33
3.4 Final remarks ........................................................................ 33

4 Adding contextual information for spectrum-based fault localization 35
4.1 Research questions ............................................................... 35
4.2 Contextualizing fault localization ............................................ 35
    4.2.1 Code hierarchy ............................................................. 36
    4.2.2 Integration coverage-based debugging ............................... 37
    4.2.3 Filtering strategies to reduce code inspection ..................... 39
    4.2.4 Debugging with roadmaps .............................................. 40
    4.2.5 CH and ICD complexity ............................................... 41
    4.2.6 Road2Fault tool .......................................................... 42

4.3 Experimental evaluation ....................................................... 44
    4.3.1 Subject programs ......................................................... 44
    4.3.2 Treatments ................................................................. 45
    4.3.3 Procedure ............................................................... 46

4.4 Results and discussion ........................................................ 47
    4.4.1 RQ1: Are roadmaps combined with filtering strategies effective for fault localization? ................................. 47
    4.4.2 RQ2: Which is the best level \( l \) value to use in the Level Score filtering strategy? .......................... 51
    4.4.3 RQ3: Which is the most helpful filtering strategy for fault localization? ............................................. 52
    4.4.4 RQ4: How many methods are inspected to reach the faults using roadmaps and filtering strategies? .... 52
    4.4.5 RQ5: Are roadmaps combined with filtering strategies effective for programs containing multiple faults? ......................................................... 53

4.5 Threats to validity ................................................................ 55
4.6 Final remarks ........................................................................ 56

5 User study ................................................................................. 61
5.1 Research questions ............................................................... 61
5.2 User study design ................................................................. 61
    5.2.1 Jaguar SFL tool ............................................................. 62
    5.2.2 Participants ................................................................. 64
    5.2.3 Subject programs and faults .......................................... 65
    5.2.4 Experimental group design ............................................ 66
    5.2.5 Questionnaires ............................................................ 68
    5.2.6 Participants’ allocation .................................................. 69
5.2.7 Experimental procedure ........................................... 69
5.3 Results and discussion ................................................ 70
  5.3.1 Participants’ distribution ......................................... 71
  5.3.2 RQ6: Can SFL help more developers to locate faults? .......... 72
  5.3.3 RQ7: Can SFL help developers to locate faults faster? .......... 73
  5.3.4 RQ8: Can SFL guide more developers to inspect faulty code? .... 74
  5.3.5 RQ9: Are the study participants intended to use SFL in practice? .. 76
  5.3.6 RQ10: The level of code information impacts on the fault localization perfor-
  mance? ........................................................................ 78
  5.3.7 Programming experience ........................................... 79
  5.3.8 Developers’ comments and observations ......................... 80
  5.3.9 Developers’ behavior .............................................. 80
  5.3.10 Factors in favor of the practical use of SFL ...................... 81
  5.3.11 Factors against the practical use of SFL ........................ 82
5.4 Threats to validity ....................................................... 82
5.5 Final remarks ............................................................. 83

6 Use of data-flow in spectrum-based fault localization 85
  6.1 Research questions ..................................................... 85
  6.2 Experimental evaluation .............................................. 85
    6.2.1 Subject programs ................................................ 85
    6.2.2 Procedure ....................................................... 86
  6.3 Results and discussion ................................................. 87
    6.3.1 RQ11: Which spectrum is more effective to locate faults: data- or control-flow? ... 87
    6.3.2 RQ12: What are the costs of execution associated with data- and control-flow? 91
    6.3.3 RQ13: Which ranking metric is more effective for data- and control-flow spectra? 92
  6.4 Threats to validity ....................................................... 93
  6.5 Final remarks ............................................................. 94

7 Conclusions 97
  7.1 Findings and implications ............................................ 97
  7.2 Future work ............................................................. 99

A User study: Questionnaires, training, instructions, and consent letter 101
  A.1 Pre-task questionnaire .............................................. 101
  A.2 Post-task questionnaire ............................................. 102
  A.3 Training - About the Jaguar tool ................................ 106
  A.4 Experiment instructions ........................................... 108
  A.5 Consent letter ........................................................ 109

Bibliography 113
List of Figures

2.1 Moth found in the MARK II computer (Source: Smithsonian Institute – The National Museum of American History) .................................................. 9
2.2 Code of the max program ........................................................................ 11
2.3 Control-flow graph of the max program .................................................. 12
2.4 Data-flow graph of the max program ....................................................... 12
2.5 Code of the max program ........................................................................ 21

3.1 Fault localization topics ........................................................................... 24
3.2 Spectrum-based fault localization concept map ......................................... 34

4.1 MethodCallPair (MCP) ........................................................................... 38
4.2 Code excerpt with the Ant’s PJH_AK_1 bug in line 112 ............................ 41
4.3 Steps to create the CH and ICD roadmaps .............................................. 44
4.4 Effectiveness of treatments using various budgets for Ochiai ................. 48
4.5 Effectiveness of treatments using various budgets for Tarantula ............. 49
4.6 Faults found by ICD for different effort budgets and LS values using Ochiai 52
4.7 Faults found by CH for different effort budgets and LS values using Ochiai 53
4.8 Faults found by ICD for different effort budgets and LS values using Tarantula 54
4.9 Faults found by CH for different effort budgets and LS values using Tarantula 55
4.10 Effectiveness of methods for Ochiai ....................................................... 56
4.11 Effectiveness of methods for Tarantula ................................................... 57
4.12 Effectiveness of treatments for all multiple-fault versions using various budgets for Ochiai ................................................................. 58
4.13 Effectiveness of treatments for 2-fault versions using various budgets for Ochiai 58
4.14 Effectiveness of treatments for all 4-fault versions using various budgets for Ochiai 59
4.15 Effectiveness of treatments for all multiple-fault versions using various budgets for Tarantula ................................................................. 59
4.16 Effectiveness of treatments for 2-fault versions using various budgets for Tarantula 60
4.17 Effectiveness of treatments for 4-fault versions using various budgets for Tarantula 60

5.1 Jaguar’s architecture .............................................................................. 62
5.2 Excerpt of a Jaguar’s XML report file .................................................... 63
5.3 Jaguar SFL tool with a method list ........................................................ 64
5.4 Jaguar’s lists of lines, methods, and duas .............................................. 65
5.5 Jaguar’s lists of code hierarchy with lines and duas ................................ 66
5.6 Jaguar's roadmap lists with lines and duas ........................................... 67
5.7 Experimental procedure ............................................................................ 70
5.8 Participants’ allocation .............................................................................. 71
5.9 Participants’ distribution using Jaguar ....................................................... 72
5.10 Efficiency of Jaguar and Eclipse .............................................................. 74
5.11 Usefulness results ..................................................................................... 77
5.12 Ease of use results ..................................................................................... 78
5.13 Behavioral intention of use results ............................................................ 79

6.1 Effectiveness of dua and line spectra for Ochiai and Tarantula ............... 88
6.2 Effectiveness of dua and line spectra for Jaccard and Zoltar .................. 88
6.3 Effectiveness of dua and line spectra for DRT and Kulczynski2 .......... 89
6.4 Effectiveness of dua and line spectra for McCon and Wong3 ............... 89
6.5 Effectiveness of dua and line spectra for $O^p$ and Minus .................... 90
6.6 Effectiveness of all ranking metrics for dua spectrum ......................... 92
6.7 Effectiveness of all ranking metrics for line spectrum ......................... 93
6.8 Effectiveness of all ranking metrics for dua spectrum ......................... 94
6.9 Effectiveness of all ranking metrics for line spectrum ......................... 95

A.1 Eclipse’s Jaguar plugin .............................................................................. 106
A.2 Jaguar view ............................................................................................... 107
List of Tables

2.1 Data- and control-flow spectra for the max program ........................................... 13
2.2 Ranking metrics for fault localization ................................................................. 19
2.3 Test suite of the max program ............................................................................. 21
2.4 Coverage and suspiciousness of the max program with the Ochiai metric .......... 22

4.1 CH roadmap of the PJH_AK_1 bug ................................................................. 42
4.2 ICD roadmap of the PJH_AK_1 bug ................................................................. 42
4.3 Block list of the PJH_AK_1 bug for parse() ..................................................... 43
4.4 Block list of the PJH_AK_1 bug for all methods .............................................. 43
4.5 Average of memory size overhead of CH and ICD compared with BL ............... 44
4.6 Characteristics of the subject programs ............................................................. 45
4.7 Multiple-fault versions ...................................................................................... 47
4.8 Median of the ranking position of the bugs using Ochiai .................................. 49
4.9 Median of the ranking position of the bugs using Tarantula .............................. 50
4.10 Test hypotheses for ST1 .................................................................................. 50
4.11 Budgets with statistical significance for ST1 ................................................... 50
4.12 Test hypotheses for ST2 .................................................................................. 50
4.13 P-values (%) for ST2 ..................................................................................... 51
4.14 Test hypotheses for ST1 .................................................................................. 54
4.15 Test hypotheses for ST2 .................................................................................. 54
4.16 Budgets with statistical significance for ST1 for multiple-fault versions .......... 55
4.17 P-values (%) for ST2 for multiple-fault versions .............................................. 56

5.1 Experimental groups (G) .................................................................................... 68
5.2 Items of TAM .................................................................................................... 69
5.3 Distribution of participants through the faults ..................................................... 71
5.4 Distribution of participants through the order of tasks ....................................... 72
5.5 Statistical results for RQ6 ................................................................................ 72
5.6 Effectiveness of participants by technique and project .................................... 73
5.7 Statistical results for RQ7 ................................................................................ 74
5.8 Crosstab for clicking on the faulty line at least once and debugging result .......... 75
5.9 Crosstab of those who clicked on the faulty line at least once using Jaguar and Eclipse 75
5.10 Statistical results for RQ8 ................................................................................ 75
5.11 Participants who found the faulty methods ...................................................... 76
5.12 Factorial analysis of the TAM items ............................................................... 76
5.13 Correlation among TAM constructs ................................................................. 76
5.14 Statistical results for RQ10 .............................................................................. 78
5.15 Participants experience in years ...................................................................... 79

6.1 Characteristics of the subject programs ............................................................. 86
6.2 Statistical results .............................................................................................. 90
6.3 Data- and control-flow efficiency for each program ......................................... 91
Software development has become increasingly challenging through the years. Systems are more complex, interacting with several libraries and frameworks to perform a varied number of tasks previously done by humans. Most of the business areas all over the world are dependent on software systems. Thus, the development of reliable software is essential to avoid the occurrence of losses and accidents caused by software failures. Software failures cost around 60 billion dollars annually in the United States economy [Tassey, 2002]. Moreover, these failures can even lead to fatal accidents, especially in critical systems from areas such as aeronautics, medicine, or astronautics [Wong et al., 2017].

Software testing and debugging are two essential tasks to ensure software quality. Since testing reveals a failure, a developer can inspect the code to identify possible causes that lead to this incorrect behavior.

Debugging is one of the most time-consuming activities in software development [Hailpern and Santhanam, 2002]. Debugging is a two-step process. It comprises locating and fixing bugs [Myers, 1979]. In practice, debugging is a manual task [Böhme et al., 2017; Kochhar et al., 2016]. Hailpern and Santhanam [2002] consider that debugging is still very much an art. Developers often use symbolic debuggers and print statements to search for faults. Since these manual processes can be expensive and ad-hoc [Jones et al., 2007], approaches that automate fault localization are valuable to reduce costs in the software development process.

Several fault localization techniques have been proposed in the last decades. A promising approach, called Spectrum-based Fault localization (SFL) [Abreu et al., 2007], consists of techniques that use information from test execution to provide a list of suspicious program elements (e.g., statements, basic blocks, methods) more likely to be faulty [Abreu et al., 2007; Jones et al., 2002; Renieris and Reiss, 2003]. Developers should inspect a suspiciousness list to search for faults. However, several limitations are hampering the adoption of fault localization techniques in industrial settings.

In this thesis, we investigate factors that can improve the use of SFL techniques in practice. We propose the use of contextual information—by combining lists of suspicious methods and their respective basic blocks with filtering strategies—to improve the effectiveness of SFL and to bring more information for helping developers to find bugs. We also assess how developers use SFL in practice and whether they consider using SFL for their daily activities. Finally, we evaluate the use of data-flow spectrum for SFL.

Our results show that the use of contextual information improves the effectiveness of SFL by reducing the amount of code to inspect until finding bugs. A few methods need to be inspected to reach most of the faults. Our findings also indicate that SFL can be used in practice since it helps developers to find and to get closer to faulty code excerpts. In our user study, SFL is well accepted by the participants, who showed intention to use it in their daily activities. Also, our results indicate that data-flow spectrum improves the fault localization effectiveness of SFL, which means that more faults were well-ranked using data-flow than control-flow spectrum. The cost of

\[\text{footnote}{1}\text{We use the terms entity, element, component, and code excerpt interchangeably throughout this thesis.}\]
execution of data-flow is higher than control-flow, but it is not prohibitive. The time to create the data-flow SFL lists varied from 22 seconds to 8:25 minutes in the worst case for the large-sized programs used in the evaluation.

SFL techniques should not only rank the faulty code among the first picks, but they should provide more information to help developers in their debugging activities. Hereupon, providing information about suspicious methods and variables can improve the developers’ experience using SFL. Our user study indicates that even when the faulty lines are not well-ranked, SFL can lead developers close to faulty code regions, which helps to locate such bugs. Although more user studies are needed to better understand SFL in practice, SFL techniques should be developed for use in real applications and disclosed to the software community.

1.1 Context and problem outline

SFL techniques use ranking metrics to calculate the suspiciousness values of program elements. Several ranking metrics have been used for this purpose [Abreu et al., 2009a; Jones et al., 2002; Naish et al., 2011]. Some studies have compared the existing ranking metrics, showing that most of them present similar performance [Debroy and Wong, 2011; Le et al., 2013a; Pearson et al., 2017; Xie et al., 2013]. To cope with this “ranking metric barrier”, there is a need for new approaches that aim to improve the SFL results.

Moreover, SFL techniques are assessed based on assumptions about the developers’ behavior when inspecting suspiciousness lists. A developer is supposed to inspect an SFL list from the most to the least suspicious program elements until reaching the faulty one. This assumption leads to some implications: first, the techniques are assessed only by the position of a bug in a list; second, a bug is deemed as found when the faulty element is reached. This way of assessment is useful to compare the effectiveness of different techniques, but it may not be useful to assess the performance of SFL in practice. The study of Parnin and Orso [2011] showed that (1) the position of a bug in an SFL list is not enough to find it; and (2) reaching a bug in an SFL list does not guarantee that it will be found.

A developer using an SFL technique may start by investigating the top-ranked statements. If s/he does not find the bug after investigating the top one, the next statement may be verified. Current SFL techniques lack contextual information. Suppose a developer is looking for faults in a large program, in which several code excerpts have the same suspiciousness score. If s/he had to choose one to initiate the investigation, which piece of code should s/he investigate first? If the bug is not found in the first pick, which ones should be investigated next? Moreover, a typical ranking list, sorted only by suspiciousness, may comprise scattered statements—a statement at the top of the list followed by a statement that belongs to a completely different site in the source code, requiring a large cognitive effort from the developer to jump from one context to the other. Thus, providing more contextual information about the places where the code elements are and how they are related in the code may help to improve fault localization.

Despite the importance of improvements in SFL techniques to achieve better results, few studies have tried to understand whether and how developers will use such results in practice [Gouveia et al., 2013; Parnin and Orso, 2011; Xia et al., 2016; Xie et al., 2016]. Moreover, these studies have presented divergent results, which indicates the need for more user studies to further increase knowledge on how developers actually use SFL. Such knowledge can lead to new techniques for use in real settings. Replication studies play an important role to improve the practice of software engineering, helping to understand whether previous results hold and in which conditions [Shull et al., 2008].

Since debugging is a complex and intellectual task, it is too simplistic to assume that SFL can replace the developer’s role in locating bugs. SFL can be useful to provide hints and to guide developers closer to the faults. A recent survey of Kochhar et al. [2016] asked developers for their expectations on fault localization techniques. Regarding the level of suspicious information, participants deemed methods as the preferred level. Indeed, methods enclose the logic of a program’s
functionalities, which may be easier to comprehend than isolated statements. Also in Kochhar et al. [2016], most participants answered that they are willing to adopt an accurate and efficient fault localization technique.

Another open issue of the SFL area is the type of spectrum used by the current techniques. Most SFL techniques use only control-flow information, especially statement and branch spectra [Abreu et al., 2007; Chilimbi et al., 2009; Jones et al., 2007; Renieris and Reiss, 2003]. Data-flow spectrum can provide more detailed information for fault localization since it exercises more relationships during the program execution. Indeed, previous studies have shown that data-flow spectrum can improve fault localization [Masri, 2010; Santelices et al., 2009]. These studies, though, used only small- to medium-sized programs in their assessments.

The reason why few SFL studies use data-flow is due to the cost to gather this data. To obtain the spectra for fault localization, the original source or object code needs to be modified (instrumented) to track the elements executed by each test case. The cost of data-flow spectrum (e.g., definition-use associations or duas as defined in Section 2.2.1) is higher because it comprises all value assignments (variable definition) and their subsequent references (variable uses) in a program [Hutchins et al., 1994; Santelices et al., 2009]. As a result, there are much more duas to track at run-time than statements, which implies both execution and memory overheads during test case execution. However, recent results have shown that dua spectrum can be obtained with fairly moderate execution and memory overheads [Chaim and de Araujo, 2013; de Araujo and Chaim, 2014]. These results motivate us to investigate the use of data-flow spectrum for SFL. Moreover, data-flow spectra such as duas provide information about uses and definitions of variables, which can provide more context to understand faulty behaviors when searching for bugs.

Thus, there are several challenges to improve SFL aiming at its practical use. Techniques should propose new ways to improve the fault localization effectiveness. They should also provide more information to help developers to locate bugs. Notwithstanding, it is important to understand how developers use SFL to create techniques that are useful in real settings.

1.2 Objectives

The goal of this research project is to tackle different aspects regarding the improvement of SFL for its practical adoption. First, we propose new ways to improve the fault localization effectiveness by reducing the amount of code to be inspected to locate faults. Second, we assessed how developers use SFL. Third, we evaluate the effectiveness of data-flow spectrum for SFL.

In the first part, we aim to answer the following general question:

Can contextual information improve SFL effectiveness?

Most SFL techniques provide information at the statement-level, block-level, or branch-level for code inspection [Jones et al., 2002]. A few techniques provide only method-level information [Kochhar et al., 2016; Mariani et al., 2011]. We combined both method- and block-level information to add more context to fault localization, providing a roadmap for code inspection [de Souza and Chaim, 2013]. Blocks are grouped by their respective methods. We propose to combine roadmaps with filtering strategies to reduce the number of blocks to be inspected in each method. Thus, we assess the following research questions:

RQ1: Are roadmaps combined with filtering strategies effective for fault localization?

To answer this question, we carried out an experiment using 62 faults from 7 real open source projects. We compared the roadmaps and filtering strategies with a list of basic blocks, which represents a single-level list presented by other SFL techniques.

RQ2: Which is the best level ($l$) value to use in the Level Score filtering strategy?
One of the filtering strategies, called Level Score, can be set with different levels of suspiciousness, providing different amounts of blocks to inspect in each method. In this question, we evaluate whether a level can reach more faults without increasing excessively the number of blocks to inspect.

**RQ3: Which is the most helpful filtering strategy for fault localization?**

In this question, we compare the performance of our filtering strategies to assess whether one of them has better effectiveness.

**RQ4: How many methods are inspected to reach the faults using roadmaps and filtering strategies?**

This question evaluates the fault localization effectiveness from the method-level perspective. The knowledge of the number of methods that reach most of the faults may impact on the decision of using an SFL technique.

**RQ5: Are roadmaps combined with filtering strategies effective for programs containing multiple faults?**

A program contains an unknown number of faults. Thus, it is important to evaluate how an SFL technique behaves in the presence of multiple faults. We combined faults from six of the seven programs previously used to create 2-fault and 4-fault versions to assess this question.

For the second part of this research project, we want to answer the following question: **Is SFL useful in practice?**

SFL techniques have presented promising results to help developers in theoretical assessments. However, few studies evaluate how developers use SFL in practice. These studies have shown divergent results regarding the performance of SFL. This fact motivated us to conduct a user study to evaluate the concerns of previous studies (RQ6 and RQ7) and also new concerns regarding the practical use (RQ8 and RQ9). The following research questions aim to answer these concerns:

**RQ6: Can SFL help more developers to locate faults?**

RQ6 evaluates the effectiveness of using SFL to locate bugs. Can developers benefit from SFL to locate bugs in practice? This is a fundamental question since SFL will only be adopted in practice if it improves the developers’ ability to locate faults. To answer RQ6, we compare the number of developers who found bugs using SFL and without using SFL.

**RQ7: Can SFL help developers to locate faults faster?**

RQ7 evaluates the efficiency of SFL in locating bugs. A technique that reduces time spent in debugging may encourage developers to use it. To answer RQ7, we compare the time spent by developers who found bugs using SFL and without using it.

**RQ8: Can SFL guide more developers to inspect faulty code?**

RQ8 evaluates whether SFL guides developers towards faulty code excerpts. By doing so, developers will be more prone to inspect the code for faults where they occur. Reaching faulty sites may be the first step to understand a bug. To answer RQ8, we verify whether SFL leads more developers to inspect the faulty line and to locate the faulty methods.

**RQ9: Are the study participants intended to use SFL in practice?**
In RQ9, we investigate whether the participants of our user study are interested in using SFL in their daily activities. This is another important question regarding the practical adoption of SFL. Developing and proposing new automated techniques to aid debugging is worthy only if developers intend to use them. To answer this question, we use the Technology Acceptance Model (TAM) to ask developers how they evaluate usefulness and ease of use of the SFL tool and if they intend to use SFL in the future.

**RQ10: The level of code information impacts on the fault localization performance?**

In RQ10, we assess whether the level of code information has some impact on the developers’ ability to locate faults. To answer RQ10, we compared the fault localization results of developers using SFL lists of methods and lines.

Beyond the questions presented above, we also evaluate other important issues related to practical use of SFL. First, we investigate how programming experience impacts on the developers’ performance using SFL. Second, we verify how developers behave during debugging tasks—how they navigate through the SFL lists, and how they search for bugs. Finally, we discuss the developers’ comments and opinions about SFL.

The third part of this research project aims to answer the following research questions:

**Can data-flow spectrum improve SFL?**

Despite its ability to distinguish more code execution relationships than control-flow, data-flow spectrum has not been used by SFL techniques. We compared the performance of data- and control-flow for SFL using 163 faults from 5 real open source programs. For data-flow, we used definition-use associations (duas) as the spectrum unit. For control-flow, we used line spectrum. Our evaluation was guided by the following research questions.

**RQ11: Which spectrum is more effective to locate faults: data- or control-flow?**

A program contains more duas than lines or branches. This fact leads dua spectrum to provide more information about relationships among program elements than line spectrum. In RQ11, we assess whether data-flow spectrum improves SFL by pinpointing faults more precisely than control-flow spectrum.

**RQ12: What are the costs of execution associated with data- and control-flow?**

As data-flow provides more detailed information, a possible drawback is related to its costs of execution. In RQ12, we assess and compare the run-time overheads associated with both spectra.

**RQ13: Which ranking metric is more effective for data- and control-flow spectra?**

SFL techniques use ranking metrics to calculate the suspiciousness of program entities. Several ranking metrics have been used for this purpose, and SFL studies have presented different results for control-flow spectrum depending on the programs used in the experiments. In RQ13, we assess and compare the effectiveness of ten ranking metrics for data- and control-flow spectra on five real open source projects.

1.3 Contributions

The original contributions of this thesis are:

- Scientific contributions:
1. New approaches to contextualize and improve fault localization, which combine two previous roadmap-based techniques with two new filtering strategies. We evaluated the contextualization techniques in programs of various sizes with real and seeded bugs, and containing single and multiple bugs.

2. An analysis of how SFL helps developers locate faults. We assessed the debugging performance of 26 developers with and without SFL using two faults from real programs. In this study, we also evaluated the developers’ intention to use SFL in the future and their behavior while using SFL.

3. A study comparing the use of data-flow and control-flow spectra for fault localization. We assessed both spectra in programs of various sizes with real bugs. We also evaluated the performance of data- and control-flow using ten ranking metrics.

4. A survey on SFL techniques from 2005 to October 2017 that describes the main advances and challenges of this research area. The survey presents a concept map relating the sub-areas of SFL studies.

5. Provision of new faults for experimentation from an open source Java program called jsoup. This program is available along with other programs in a source code repository for experimental purposes, and it has been used by the research community [Li et al., 2016; Ribeiro, 2016].

6. A case study of the method (called haphazard intentional allocation – HIA) that we used to automatically allocate participants in the user study, as described in Section 5.2.6.

- Technological contributions:

7. The Jaguar (JAva coveraGe faUlt locAlization Rank) open source SFL tool, which is available for use in real settings by researchers and practitioners. Jaguar implements data- and control-flow spectra. It is available as an Eclipse plug-in and as a command line tool.

8. The Road2Fault open source SFL tool, which implements MCP-, data- and control-flow spectra. Road2Fault is available as a command line tool.

1.3.1 Publications

We wrote the following papers during this doctoral research:

Published papers:


To be submitted in the first semester of 2018:


Tool demonstration:


The publications listed below are indirectly related to this research project:

Published papers:


Submitted papers:


1.3.2 Other contributions

• Jaguar: github.com/saeg/jaguar

• Road2Fault: github.com/saeg/road2fault

• Programs for experimentation: github.com/saeg/experiments/tree/master/jaguar-2015

• Experimental packages:
  – Contextual information study: github.com/saeg/road2fault
  – User study: github.com/saeg/user-study-sfl
  – Data-flow study: github.com/saeg/data-flow-sfl

1.4 Structure of the thesis

The remainder of this thesis is organized as follows:

• Chapter 2: presents basic concepts related to this work, which includes debugging, fault localization, testing, contextual information, user studies, and filtering strategies;
• Chapter 3: describes the survey we conducted, which includes a historical overview of fault localization, the main advances and challenges, and related work;

• Chapter 4: presents our study about the use of contextual information for SFL;

• Chapter 5: details our user study;

• Chapter 6: presents the study about the use of data-flow spectrum for SFL;

• Chapter 7: presents our conclusions and future work;

• Appendix A: details our user study’s materials, which includes questionnaires, training material, experiment’s instructions, and consent letter.
Chapter 2

Basic concepts

This chapter presents concepts related to debugging, fault localization, and testing, which are addressed throughout this thesis. We also present a running example of spectrum-based fault localization.

2.1 Debugging

Program faults are an inevitable consequence of writing code. Faults occur for various reasons: typing errors, misunderstanding of software requirements, wrong assignments to variables, or absence of code to verify some unpredicted condition. The use of the word *bug* originates in Thomas Edison’s time. It was used to indicate flaws in engineering systems [Kidwell, 1998]. In the late 1940s, the *Mark II* computer at *Harvard University* suddenly stopped. Technicians found that a dead moth had shorted out some of the computer’s circuits, and taped the bug into the machine’s logbook [Kidwell, 1998] (see Fig. 2.1). Since then, the term *bug* has been used to refer to problems in computer systems. The term *debug* was then associated with the activities of finding and fixing program faults.

![Moth found in the MARK II computer (Source: Smithsonian Institute – The National Museum of American History)](image)

During the testing phase or in the field, a fault is revealed when a program presents an unexpected behavior (known as failure). Once a failure occurs, the two-step debugging process begins. First, a developer inspects the code to locate the failure’s cause. Second, s/he fixes the fault [Myers,
Debugging is also a cognitive-demanding problem-solving process, which requires knowledge and reasoning skills to understand a program and then to locate and fix bugs [do Yoon and Garcia, 1998].

2.1.1 Fault, failure, error, and mistake

Faults, errors, and failures represent three stages in a program’s execution during which an unexpected behavior occurs. The IEEE Standard 610.12 [IEEE, 1990] defines fault, error, and failure as follows. Fault is an incorrect step, process, or data definition in a computer program. A fault is inserted by a developer who wrote the program. A fault is also called bug or defect. However, a fault is a subtype of a defect, since a defect can also be detected by inspection, prior to the software execution [IEEE, 2009].

Error is a tricky term, which is also sometimes used to refer to a fault, failure, or mistake. In its particular sense, an error is a difference between a computed value and the correct value [IEEE, 1990]. The term is often used to indicate an incorrect state during the program execution. Thus, an error occurs when an executed fault changes the program state. Other terms used to express error are infection [Zeller, 2009] and anomaly.

Failure describes a system’s inability to perform its function at the expected requirements [IEEE, 1990]. A failure is observed as an unexpected output, which occurs when an error in a program state leads to a wrong output. A synonym for failure is unexpected behavior. Crashes are failures that interrupt program executions and thus have an apparent behavior. Finally, a mistake is a human action that produces a fault [IEEE, 1990].

2.1.2 Program comprehension

Program comprehension (or program understanding) is known as descriptions of what programs do and how they do it [Lukey, 1980]. Developers must understand programs to perform their programming and debugging activities, which are considered as complex tasks, especially for unfamiliar programs [Letovsky, 1987]. According to Biggerstaff et al. [1993], a person comprehends a program when s/he can explain it: its structure, its behavior, and its relationships with its application domain. Lukey [1980] argues that programs are often large to be understood in their entirety as a single unit. Thus, segmenting programs into code chunks that form units of analysis can improve the program understanding process. Methods or functions should represent such units of analysis.

Moreover, program comprehension entails other behavioral factors, such as cognitive processes from microscale (e.g., stretching from eye fixation) to macroscale (e.g., decision taking, problem-solving) [Letovsky, 1987]. Developers should also study source code and documentation, and talk with other developers and requirements engineers to understand programs. Gould [1975] stated that debugging is much more complex than many other systems that are diagnosed (e.g., automotive, human anatomical, or plumbing). It occurs due to the variety in which programs can be written and diagnosed. According to Corbi [1989], program comprehension is a key activity in software development, and developers spend most of their time comprehending code.

2.2 Testing

Software testing is a fundamental step for debugging since it reveals the existence of failures in programs [Myers, 1979]. Automated software testing allows the reproduction of failures, which facilitates the inspection for faults that are responsible for these failures. As stated by Dijkstra [1972], “Program testing can be used to show the presence of bugs, but never their absence”. Testing has been used in the software industry to improve quality assurance [Bertolino, 2007].

A Test case is composed of input values and an output (expected result) that are necessary to complete some execution of a program under test [Offutt and Anmmann, 2008]. The expected result is the intended behavior of a program for the input values.
Testing oracle or simply oracle is an entity responsible for defining a correct output for some test case input, which allows verifying whether the test execution result is correct. Generally, an oracle is defined by someone with knowledge about what a program should present as a valid output.

A passing test case is one that satisfies the expected behavior. Conversely, a failing test case does not satisfy its corresponding expected behavior. Synonyms for passing are successful, correct, and passed. Failing synonyms are failure, unsuccessful, incorrect, and failed. Test cases are also called runs, executions, traces, and tests.

### 2.2.1 Testing strategies

Testing strategies aim to test software under different perspectives. The most common strategies are structural testing, functional testing, and fault-based testing.

#### Structural testing

Structural testing is a strategy that uses the internal structures of a program to create the tests [Offutt and Ammann, 2008]. It is also known as white-box testing. This strategy aims to test software from the developers’ perspective to make sure that the code structures are being covered. Information gathering from these code structures is known as code coverage or program spectrum and can be composed of any granularity of a program, from a statement to a subprogram. These code structures are also known as program elements, components, or program entities. The criteria for structural testing are based on control-flow, which investigate paths executed by the tests, or data-flow, which analyze definitions and uses of variables through the executions. Program entities include statements, basic blocks, branches, predicates, definition-use associations, variables, components, functions, methods, modules, and so on.

The source code structure of a program is represented by a control-flow graph (CFG), in which nodes are basic blocks and edges are all possible branches that link blocks. A basic block is a set of statements executed sequentially in which all block statements are executed once the first block statement is executed [Hennessy and Patterson, 2011]. Figure 2.3 shows the control-flow graph of the max program (Figure 2.2), which is used in our running example in Section 2.4.6.

<table>
<thead>
<tr>
<th>Line</th>
<th>Statement</th>
<th>Block</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>int max(int[] array, int length)</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>1</td>
<td>{</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>int i = 0;</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>1</td>
<td>int max = array[++i]; // array[i++];</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>2</td>
<td>while(i &lt; length)</td>
</tr>
<tr>
<td>6</td>
<td>-</td>
<td>3</td>
<td>{</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>3</td>
<td>if(array[i] &gt; max)</td>
</tr>
<tr>
<td>8</td>
<td>5</td>
<td>4</td>
<td>max = array[i];</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>5</td>
<td>i++;</td>
</tr>
<tr>
<td>10</td>
<td>-</td>
<td>5</td>
<td>}</td>
</tr>
<tr>
<td>11</td>
<td>7</td>
<td>6</td>
<td>return max;</td>
</tr>
<tr>
<td>12</td>
<td>-</td>
<td>6</td>
<td>}</td>
</tr>
</tbody>
</table>

Figure 2.2: Code of the max program

A CFG can be extended to include data-flow information, which is known as data-flow graph (DFG). Figure 2.4 shows the data-flow graph of the max program. A DFG indicates the definitions and uses of variables in a program. A definition is a location where a variable receives a value, i.e., stored in memory [Offutt and Ammann, 2008]. A use is a location where a value in a variable is accessed [Offutt and Ammann, 2008]. A variable use is deemed as a computation use (c-use) when it affects a computation being performed or it allows to see the result of a previous definition.
Figure 2.3: Control-flow graph of the max program

Otherwise, a use is deemed as a *predicate-use* (p-use) when it affects the control-flow execution [Rapps and Weyuker, 1985].

Data-flow coverage can be formally defined using the concept of *definition-use associations* (duas) [Rapps and Weyuker, 1985]. The triple \( D = (d, u, X) \), called c-use dua, represents a data-flow testing requirement involving a definition in node \( d \) and a c-use in node \( u \) of variable \( X \) such that there is a *definition-clear path* with respect to \( X \) from \( d \) to \( u \). A definition-clear path with respect to a variable \( X \) is a path where \( X \) is not redefined in any node in the path, except possibly in the first and last ones. Likewise, the triple \( D = (d, (u', u), X) \), called p-use dua, represents the association between a definition and a p-use of a variable \( X \). In this case, a definition-clear path \((d, \ldots, u', u)\) with respect to \( X \) should exist.

![Data-flow graph of the max program](image)

Figure 2.4: Data-flow graph of the max program

The CFG and DFG are used to define the structural testing criteria. Two common control-flow criteria are *all-nodes* and *all-edges*, while two well-known data-flow criteria are *all-defs* and *all-uses*. The definition of these criteria are presented below:

**All-nodes**: all \( n \) nodes of a program \( P \) must be covered at least once by the test suite \( T \).

**All-edges**: all \( e \) edges of a program \( P \) must be covered at least once by the test suite \( T \). This criterion includes the All-nodes criterion since covering all edges implies covering all nodes.

**All-defs**: a definition-clear path for some dua \((d, u, X)\) or \((d, (u', u), X)\) for each definition \( d \) of
a program \( P \) must be covered at least once by the test suite \( T \). This criterion may lead to different sets of duas since a definition can be used more than once.

**All-uses**: a definition-clear path for each dua \((d, u, X)\) or \((d, (u', u), X)\) of a program \( P \) must be covered at least once by the test suite \( T \). This criterion subsumes the All-defs criterion since it contains any chosen dua in All-defs.

Table 2.1 contains the program elements that meet the criteria defined above for the \( max \) program. These elements are the testing requirements of each criterion. Through the example of the \( max \) program, we can see why data-flow is more expensive since it leads to a higher number of testing requirements from the source code.

**Table 2.1: Data- and control-flow spectra for the max program**

<table>
<thead>
<tr>
<th>All-nodes</th>
<th>All-edges</th>
<th>All-defs</th>
<th>All-uses</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(1, 2)</td>
<td>(1, 6, max)</td>
<td>(4, 6, max)</td>
</tr>
<tr>
<td>2</td>
<td>(2, 3)</td>
<td>(1, (3, 4), max)</td>
<td>(1, (3, 5), max)</td>
</tr>
<tr>
<td>3</td>
<td>(2, 6)</td>
<td>(1, (2, 3), length)</td>
<td>(1, 4, i)</td>
</tr>
<tr>
<td>4</td>
<td>(3, 4)</td>
<td>(1, (2, 3), i)</td>
<td>(1, (2, 6), i)</td>
</tr>
<tr>
<td>5</td>
<td>(3, 5)</td>
<td>(4, 6, max)</td>
<td>(5, 4, i)</td>
</tr>
<tr>
<td>6</td>
<td>(4, 5)</td>
<td>(5, (2, 3), i)</td>
<td>(5, (2, 6), i)</td>
</tr>
</tbody>
</table>

**Functional testing**

Functional testing is a strategy in which the tests cases are created using specifications of a program [Myers, 1979]. Only the inputs and expected outputs are used in this strategy. That is why functional testing is also known as black-box testing (i.e., without knowledge of the internal structures of a program). This strategy is more related to a customers’ perspective, which aims to verify if the functionalities of a program are being met.

A possible way to perform this strategy is the exhaustive input testing criterion [Myers, 1979] to create test cases for all possible input conditions. Of course, this approach is unfeasible to be applied in most cases. Thus, there are other non-exhaustive criteria used in functional testing, such as random testing, equivalent class partition, and boundary value analysis [Burnstein, 2003]:

**Random testing**: input values are randomly selected from a range of possible values. Although this criterion may be less costly, requiring less planning to execute, the chances of obtaining adequate test cases are also random.

**Equivalent class partition**: the input domain is divided into equivalent classes. These classes are identified from sets of values that present an equivalent behavior in the domain. Thus, only one value from each partition needs to be tested. This criterion reduces the number of input values and assures that different behaviors will be covered.

**Boundary value analysis**: this criterion explores the boundary values of the input domain. It is used along with the equivalent class partition criterion to test how a program behaves using these boundary values.

**Fault-based testing**

Fault-based testing is a strategy in which faults are put into the source code to reveal failure behaviors. Faulty versions of a program under test are created. These faulty versions should be
revealed by the test cases. Thus, this strategy is used to reveal faults and to assess test suite quality [Burnstein, 2003]. The most famous fault-based testing criterion is mutation testing.

**Mutation testing**: a mutant is a faulty version of a program that contains one or more faults. A fault is inserted using mutant operators at some statement of interest. There are several mutant operators, each of them related to specific fault types (e.g., statement deletion, arithmetic operation replacement, logical operator replacement). A mutant that leads to a different output value is deemed as killed by the test suite. Test suites that are not capable of killing mutants should be improved with additional test cases, or analyzed for possible equivalent mutants.

Besides its use for the improvement of test suite quality, mutation testing has been used for other activities. For example, mutant testing is used to create faulty versions of programs for experimentation in testing and debugging [Do et al., 2005]. It is also used to suggest possible fixes for program repair [Debroy and Wong, 2010] and to propose new fault localization techniques [Moon et al., 2014; Papadakis and Le Traon, 2014].

### 2.3 Debugging techniques

Different tools and techniques have been proposed to support debugging. Most of them allow developers to inspect program states by themselves, while several approaches have tried to provide information about suspicious code excerpts to reduce the effort needed to identify bugs.

Due to the diversity of studies on fault localization, several terms have been used to define similar concepts. In what follows, we clarify terms and synonyms used in this thesis.

#### 2.3.1 Symbolic debuggers

Most of the debugging tools used by developers today were proposed in the 1960s [Agrawal and Spafford, 1989], and earlier debugging tools originate from the late 1950s [Evans and Darley, 1966]. These tools are the first versions of the **symbolic debuggers**, which allow developers to check the state of program executions to search for faults. Some examples are Gilmore’s debugging tool (called *TX-O Direct Input Utility System*) [Gilmore, 1957], FLex Interrogation Tape (FLIT) [Stockham and Dennis, 1960], and DEC (Digital Equipment Corporation) Debugging Tape (DDT) [Kotok, 1961]. The TX-O Direct Input Utility System influenced subsequent, more sophisticated debugging tools. It was based on the idea of moving the program under debugging to the computer’s memory, making it possible to verify and modify registers during the execution.

**Breakpoints** are used to inspect a program’s source at some point of interest. As the first tool to implement the concept of a breakpoint, FLIT also allows modifying the program during its execution. DDT evolved from FLIT for the PDP-1 computer. Another advance in the 1960s debugging tools was the conditional breakpoint, which permits a breakpoint’s execution only when it reaches some specific condition. The first tools to provide code tracing were those for high-level languages, such as debugging tools for Lisp and Prolog. Beyond code tracing, debugging tools for high-level languages do not present any additional features compared to those for assembly [Evans and Darley, 1966].

Backward and forward navigation, information of control-flow and data-flow, and log tracking are another resources provided by debugging tools proposed in the 1960s. Balzer [1969] presented a tool called *EXtendable Debugging and Monitoring System (EXDAMS)* [Balzer, 1969], which was one of the first tools to allow either backward or forward navigation through the code. Its visualization provides control-flow and data-flow information using graphics, such as a tree structure of the execution at some point of interest. The execution data was stored in a history tape. However, EXDAMS did not use this information to suggest statements more likely to contain bugs, which would help developers in fault localization. Recent research in the field includes the design and implementation of distributed symbolic debuggers [Mega and Kon, 2004, 2007].
Indeed, the debugging tools used in industrial settings today do not differ much from the above-described techniques. Symbolic debuggers are available for most languages, and they are widely used. However, these tools do not provide any additional information to help developers to locate faults.

### 2.3.2 Automated debugging

Automated debugging, also known as Fault Localization, comprises techniques that automatically suggest code program excerpts more likely to contain faults, i.e., fault candidates [Abreu et al., 2009a]. Since the 1970s, several techniques have been proposed to help to automate the debugging process. These techniques tried to apply different approaches to support fault localization. This section presents a brief historical overview to introduce such approaches and how automated debugging techniques have evolved throughout the years.

#### Program models and specifications

The first automated techniques tried to rely on the correctness of programs or on the existence of correct specifications to indicate faults. Since a correct program exists, it would be easy to compare this program to other versions to identify bugs. For example, Nagy and Pennebaker [1974] proposed one of the earliest techniques to automatically identify bugs by comparing successive versions of a program, considering that the changes in code are bug corrections. Thus, their technique provides bug patterns to aid debugging.

Some previous techniques tried to understand a program’s whole behavior. These techniques depend on correct specifications of programs, which in practice are very difficult to obtain. Adam and Laurent [1980] proposed a tool called LAURA. The tool receives a program model represented by graphs. To identify faults, LAURA makes program transformations to compare the original program with the program model. Johnson and Soloway [1985] proposed a tool called PROgram UnderSTanding (PROUST), which receives programming plans, the intentions that a developer has to write the code, and a program. PROUST has a knowledge base of programming plans that it compares to the input plan’s goals. PROUST then generates a diagnostic output of bugs found in the code, including an explanation of mistakes causing the bugs.

#### Assertions

 Assertions are formal constraints that indicate a correct program state at some execution point Rosenblum [1995]. Fairley [1979] proposed a debugging tool called Assembly Language Assertion Drive Debugging INterpreter (ALADDIN), which uses breakpoint assertions instead of breakpoint locations. The breakpoint assertion is executed when a wrong value occurs in the program state at a certain point. Assertions can be helpful for detecting errors in some circumstances, especially for functions that calculate values or must contain a certain number of elements. However, it is not always possible to use assertions to detect all incorrect program behaviors, which may be infeasible for large and complex programs. Lukey [1980] proposed a technique called Program and Understanding Debugging SYstem (PUDSY), combining program understanding with assertions. PUDSY is used to derive descriptions of a program for understanding it. The technique can also generate fixes to problems in variable assignments that seem to be incorrect according to some patterns. For debugging, PUDSY generates assertions for known patterns of code, called schematas. PUDSY uses these assertions to find anomalies between what the code does and what the code is supposed to do.

#### User interaction

Another approach to automated debugging is the use of the developers’ knowledge to identify faults. Shapiro [1983] proposed two algorithms to detect incorrect program procedures: one for deterministic programs, and another for non-deterministic programs. These algorithms ask an oracle
if the output for a given input to a procedure is correct, repeating the process for the following program execution procedures. The first procedure with an incorrect output is deemed as the incorrect procedure. The algorithms proposed by Shapiro [1983] suppose that developers perform the role of the oracle for each executed procedure, which may be an error-prone and time-consuming activity for long-running and large programs. Shahmehri et al. [1990] propose a variation of the Shapiro’s algorithmic debugging [Shapiro, 1983] that reduces the number of interactions with users. The reduction is obtained by using program slicing.

**Program dependence**

Program dependence can be defined as control and data relationships among program entities, in which an entity depends on the execution of other entities [Ferrante et al., 1987]. Control dependences are those related to execution flows in the structure of programs. Data dependences occur among entities that define or use variables through the execution. A program dependence graph (PDG) represents sequences of both control and data dependences of programs, in which nodes are statements and predicate expressions, and edges represent data and control dependences.

Program dependences can be static or dynamic. Static dependence must consider all possible relationships among entities. Dynamic dependence only considers the relationships occurred in an execution, which implies in a small number of dependences compared to static dependences.

Koren [1988] proposed a tool called Program Error-Locating Assistant System (PELAS)—the first fault localization technique based on program dependence. PELAS asks developers about a behavior’s correctness and uses the answer to indicate possible fault locations. Program dependence data is used to guide the developer navigation in searching possible fault sites. The author states that the backtracking reasoning used (based on program dependence) is an abstraction of experienced developers’ intuitive processes.

**Program slicing**

Program slicing is a technique that reduces the amount of code for inspection by creating executable subsets of programs, called slices, that contain desirable program behaviors [Weiser, 1981]. These subsets are composed of statements that are related to a slice criterion. A slice criterion is a specific point of interest, such as a statement or a variable. Thus, a developer focuses his/her attention on a reduced part of a program to understand a program or to debug it.

Program slicing can also be viewed regarding its direction. In backward slicing, a slice is composed of statements that a slice criterion depends on. Conversely, in forward slicing, a slice contains statements that depend on a slice criterion [Zeller, 2009]. The difference between two slices is called as dice [Agrawal et al., 1995]. Dice can be used to check differences between two points of interest.

Program slicing was proposed to provide program static information [Weiser, 1981]. Korel and Laski [1988] proposed dynamic slicing to reduce slice size. Dynamic slices are composed of only executed statements. However, the remaining code is still excessive, which makes it impractical for large programs [Burger and Zeller, 2011; Xu et al., 2005]. Thus, several approaches for reducing program slicing outputs have been proposed [Jhala and Majumdar, 2005; Sridharan et al., 2007]. Despite the need for coping with large amounts of data, program slicing has been used to propose several fault localization techniques [DeMillo et al., 1996; Ju et al., 2014; Ko and Myers, 2008].

**Delta debugging**

Delta debugging is a technique to isolate failure causes by narrowing down differences between runs [Zeller, 2009]. Delta debugging uses differences between program states of a passing and a failing run to search for differences that cause failures.

Delta debugging can obtain reduced sets of failure causes. However, it is a high-cost approach since it requires several sequential runs to obtain the sets of suspicious elements [Burger and Zeller, 2008; Wong et al., 2016]. Zeller [2002] applied delta debugging to find cause-effect chains (variables and values) that are responsible for failures.
Model-based diagnosis

Model-based diagnosis (MBD) is a technique used to locate hardware faults in electronic digital circuits. MBD explores all possible interactions in a system to build an expected model through static analysis. This model is compared to real input/output observations to diagnose components that explain the wrong behavior. The diagnostic is obtained using logical inference.

Wotawa et al. [2002] used MBD for debugging software faults. MBD generates a logical model from the source code and uses logical reasoning to obtain a minimal set of statements that explain faults (i.e., statements that are logically related to faulty behaviors). Abreu et al. [2008] combined MBD and dynamic analysis information for fault localization.

Techniques that use static analysis have their advantages: they can assure that a program is fault-free by exploring all its possible interactions; they are also independent of testing. However, the performance of these techniques is tied to formal proofs of program correctness. Such proofs are infeasible in practice, even for small general purpose programs.

Information retrieval-based fault localization

Information retrieval-based fault localization (IRFL) refers to techniques that use information retrieval approaches [Le et al., 2013b; Lukins et al., 2008; Marcus et al., 2004] to extract relevant information from natural language documents of software management systems, such as bug reports and software requests. Relevant words from these bug reports are related to source code files. Then, IRFL techniques pinpoint files that are likely to contain faults. These techniques can also be used to indicate possible locations that should be extended or modified to implement new features.

Crowd debugging

Crowdsourcing is the act of an institution that outsources its functions to an undefined network of people through an open call [Howe, 2006]. Crowdsourcing has been applied in several Software Engineering activities. Crowd debugging is the term coined for techniques that use crowdsourcing to support debugging [Mao et al., 2017].

Crowd debugging techniques collect and mine crowd knowledge from Question and Answer (QA) sites (e.g., Stack Overflow1). The aim is to provide relevant information from previous problem solutions to identify new bugs [Chen and Kim, 2015; Monperrus and Maia, 2014; Mujumdar et al., 2011].

Remarks on automated debugging

Despite all advances, in practice, fault localization has changed little over time. In real development settings, the use of print statements and symbolic debuggers are still prevalent. What is the reason for such a state of affairs in debugging? Primarily, many of the techniques do not scale to programs developed in the industry. Another reason is that the techniques are not assessed in real debugging scenarios. The developer’s behavior might differ from that expected by proponents of localization techniques [Parnin and Orso, 2011].

In what follows, we present the main concepts related to spectrum-based fault localization, which is the automated debugging approach that we use in our research.

2.4 Spectrum-based fault localization

Spectrum-based fault localization (SFL) comprises a set of techniques that use information from testing execution to indicate suspicious pieces of code more likely to contain faults [Abreu et al., 2007; Jones et al., 2002]. The term spectrum refers to the program entities covered during test execution [Reps et al., 1997]. An important advantage of SFL is that program spectra can be

1 stackoverflow.com
obtained with a relatively small time and memory overheads [Abreu et al., 2007], which allows us to evaluate SFL on real and large programs.

There are several synonyms of program spectrum used in the literature, such as code coverage, testing data, dynamic information, execution trace, execution path, path profile, and execution profile. Program entities more likely to contain faults are called suspicious, suspected, candidate, and faulty elements.

SFL techniques use ranking metrics to calculate suspiciousness scores of each entity (e.g., statements). The output of an SFL technique is a list of statements sorted in descending order of suspiciousness. The more suspicious a statement is, the more likely for it to contain a fault. Thus, a developer using an SFL technique should inspect the most suspicious statements to locate a fault.

The first fault localization technique was proposed in the late 1980s [Collofello and Cousins, 1987]. Several SFL techniques have been proposed in the last decades [Abreu et al., 2007; Jones et al., 2002; Masri, 2010; Renieris and Reiss, 2003]. We detail some of the main concepts regarding SFL in the following sections.

2.4.1 Ranking metrics

Ranking metrics are a pivotal issue on spectrum-based fault localization. They are used to calculate the likelihood that program entities will be faulty. The studies on SFL use different terms to refer to ranking metrics: technique, risk evaluation formula, metric, heuristic, ranking heuristic, coefficient, and similarity coefficient. Several metrics have been proposed or used for this purpose [Abreu et al., 2007; Debroy and Wong, 2013; Jones et al., 2002; Wong et al., 2010]. The most effective metrics are characterized by identifying suspicious code excerpts from the frequency of elements exercised in passing and failing test cases. The rationale is that entities that are more executed in failing test cases are more likely to contain a fault. Otherwise, entities that are less executed in failing test cases and more present in passing test cases are less likely to be faulty.

Table 2.2 presents some of the main ranking metrics used in SFL. Tarantula [Jones et al., 2002] was one of the first metrics proposed for fault localization. Its formula is shown in Table 2.2. The spectrum coefficients \((c_{ef}, c_{nf}, c_{ep}, \text{ and } c_{np})\) represent the four possible states in which a program component \((c)\) can be in a test execution: \(c_{ef}\) indicates the number of times a program component \((c)\) is executed \((e)\) in failing \((f)\) test cases; \(c_{nf}\) is the number of times a component is not \((n)\) executed by failing \((f)\) test cases; \(c_{ep}\) is the number of times a component is executed \((e)\) by passing \((p)\) test cases; and \(c_{np}\) represents the number of times a component is not \((n)\) executed by passing \((p)\) test cases. The spectra information is stored in a coverage matrix [Baudry et al., 2006], which contains all components executed by the test cases and the execution outcomes of these test cases (see Table 2.4). Thus, Tarantula uses the coverage matrix to calculate the frequency that each component is executed in failing test cases and divides it by the frequency that this component is executed in failing and passing test cases.

Ochiai, originally applied to molecular biology, has been used in several SFL works [Abreu et al., 2007]. It is the ranking metric that has presented the best results for fault localization [Abreu et al., 2007; Le et al., 2013a]. Ochiai does not take into account statements not executed in passing runs \((c_{np})\). The square root also reduces the weight of \(c_{nf}\) and \(c_{ep}\) in the suspiciousness score. Jaccard was used by Chen et al. [2002] to indicate faulty components in distributed applications.

The ranking metric Zoltar [Gonzalez-Sanchez, 2007] is a variation of Jaccard. Its formula increases, more than Jaccard, the suspiciousness of program entities that are more frequently executed in failing test cases. Zoltar was developed to detect errors in embedded systems using program invariants. \(O^\prime\) [Naish et al., 2011] is a ranking metric optimized for single-fault programs, Assuming the existence of a single fault in a program, only statements that are executed in all failing test cases are suspect. Wong3 [Wong et al., 2010] gradually decreases the importance of passing test cases. The idea is that the first passing test cases contribute more to the fault localization than the subsequent passing test cases.

Kulczynski2 came from the Artificial Intelligence area, and McCon was originally used in studies of plankton communities. These two ranking metrics have presented better results for programs
with two simultaneous faults [Naish et al., 2009]. DRT [Chaim et al., 2003] was proposed for use with dua spectrum in fault localization. DRT considers the number of test cases to calculate suspiciousness. Minus [Xu et al., 2013] is a Tarantula-like ranking metric that subtracts the percentage that an entity is not executed in failing test cases, and divides by the percentage that this entity is not executed for all executions. This complementary frequency is called noise; it decreases the importance of non-execution in the analysis.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tarantula</td>
<td>( \frac{c_{ef}}{c_{ef} + cn_{f}} )</td>
</tr>
<tr>
<td>Jaccard</td>
<td>( \frac{c_{ef}}{c_{ef} + cn_{f} + c_{ep}} )</td>
</tr>
<tr>
<td>OP</td>
<td>( c_{ef} - \frac{c_{ep}}{c_{ep} + c_{np} + 1} )</td>
</tr>
<tr>
<td>Kulczynski2</td>
<td>( \frac{1}{2} \left( \frac{c_{ef}}{c_{ef} + cn_{f}} + \frac{c_{ef}}{c_{ef} + c_{ep}} \right) )</td>
</tr>
<tr>
<td>McCon</td>
<td>( \frac{c_{ef}^2 - c_{ef}nc_{ep}}{(c_{ef} + cn_{f})(c_{ef} + c_{ep})} )</td>
</tr>
<tr>
<td>Ochiai</td>
<td>( \sqrt{c_{ef}} )</td>
</tr>
<tr>
<td>Zoltar</td>
<td>( \frac{c_{ef}}{c_{ef} + cn_{f} + c_{ep} + 10000 \cdot \frac{c_{np}}{c_{ef}}} )</td>
</tr>
<tr>
<td>Wong3</td>
<td>( c_{ef} - p, ) where ( p = \begin{cases} c_{ep} &amp; \text{if } c_{ep} \leq 2 \ 2 + 0.1(c_{ep} - 2) &amp; \text{if } 2 &lt; c_{ep} \leq 10 \ 2.8 + 0.001(c_{ep} - 10) &amp; \text{if } c_{ep} &gt; 10 \end{cases} )</td>
</tr>
<tr>
<td>DRT</td>
<td>( \frac{c_{ef}}{1 + \frac{c_{ep}}{</td>
</tr>
<tr>
<td>Minus</td>
<td>( \frac{c_{ep}}{c_{ep} + c_{np}} - \frac{1}{c_{ef} + cn_{f}} + \frac{1}{c_{ep} + c_{np}} )</td>
</tr>
</tbody>
</table>

### 2.4.2 Faults

In practice, we do not know how many bugs a program has; it may contain a single fault or multiple faults. Simultaneous faults may change the execution behavior, causing interferences in the test results and affecting the performance of SFL techniques [Debroy and Wong, 2009; DiGiuseppe and Jones, 2015]. Thus, it is important to perform experiments deeming the existence of multiple faults to assess how the techniques behave under such conditions.

Simultaneous faults can cause interferences in the program behavior, impacting on fault localization. Interferences among faults lead to four scenarios [DiGiuseppe and Jones, 2011]: faulty synergy, in which the faults cause more tests to fail; fault obfuscation, in which fewer tests fail due to the multiple faults; pass/fail independence, when there are no changes in the test cases’ outputs; and multi-type, in which synergy and obfuscation occurs at the same time. There are studies showing that fault interferences are frequent in multiple-fault programs [Debroy and Wong, 2009; DiGiuseppe and Jones, 2015]. Conversely, in the presence of simultaneous faults, at least one of the faults is well ranked by SFL techniques, which is known as fault prevalence [Perez et al., 2017].

Another factor to consider concerning faults is their different types, which may impact on the fault localization performance [Masri, 2010; Santelices et al., 2009; Xie et al., 2013]. Faults of code omission are difficult to locate [Xie et al., 2013; Xu et al., 2011; Zhang et al., 2009], since it is hard to automatically suggest that a statement should exist among correct code excerpts. Multi-line faults, i.e., a fault that is spread over multiple lines, are another challenging issue for fault localization techniques. Some of these faulty lines may be well-ranked while others may be ranked low, confounding the fault inspection. Also, there are faults in code excerpts that are not instrumented (e.g., global variable declarations), and thus cannot be reached by SFL techniques.

Regarding the way faults are introduced, there are two types of faults used for research in debugging: real and seeded faults. Real faults are those that naturally occur during software development. Seeded faults are those intentionally inserted for monitoring detection [IEEE, 1990]. Faults can be manually inserted for experimental purposes, or by using mutation testing [DeMillo et al., 1978]. Fault seeding is also known as fault injection.
2.4.3 Suspiciousness lists

The output of SFL techniques are lists composed of suspicious entities that are ordered by suspiciousness scores [Abreu et al., 2007; Jones et al., 2002; Wong et al., 2010]. Although suspiciousness lists are a straightforward and logical approach, there are some drawbacks to be tackled concerning their use. Most SFL techniques output lists at the lowest code level. A typical ranking list, sorted only by suspiciousness, may comprise scattered statements—a statement at the top of the list followed by a statement that belongs to a completely different site in the source code, requiring a large cognitive effort from the developer to jump from one context to the other.

Ranking lists commonly contain several program entities with the same suspiciousness scores. This fact impacts the evaluation of ranking metrics, which can vary widely. Thus, using tie-breaking strategies may help to alleviate such problem [Wong et al., 2010; Xie et al., 2010]. Fault localization techniques often assume that a developer can find a bug immediately when the faulty code in a suspiciousness list is inspected. This assumption, called perfect bug understanding, does not occur in practice [Parnin and Orso, 2011], and may mislead the fault localization performance.

SFL lists often contain all elements executed by the tests. However, studies are showing that developers may give a technique if they cannot locate a fault after inspecting the first elements of a list [Kochhar et al., 2016; Parnin and Orso, 2011]. Therefore, SFL lists tend to be more useful in practice if they pinpoint the faulty entities among the first picks.

SFL techniques have been developed focusing on reducing the percentage of code to be examined. This information is then compared to previous techniques. However, real applications comprise hundreds of thousands of lines of code; thus, even small percentages of code, let us say, 1% of 100 KLOC, imply too much code to be inspected in industrial settings. In this sense, techniques should improve the absolute position of faults to be adopted in practice [Parnin and Orso, 2011; Steimann et al., 2013].

2.4.4 Contextual information

A suspiciousness list composed of single statements may not contain information enough for a developer to understand and detect a fault [Hsu et al., 2008]. In practice, when a developer searches for a bug, s/he tries to understand the conditions in which this bug occurs. Understanding the program context that leads to the occurrence of an incorrect program state is important to generate hypotheses to fix bugs [Layman et al., 2013].

Recent techniques have been proposed to add information from the context where faults occur [Cheng et al., 2009; Hsu et al., 2008; Jiang and Su, 2007]. In fault localization, contextual information is related to ways to provide information that help developers to understand how faults occur [Jiang and Su, 2007]. Contextual information may include subgraphs comprising small execution paths [Cheng et al., 2009; Hsu et al., 2008], natural language, variable names, source code comments [DiGiuseppe and Jones, 2012], code structures [de Souza and Chaim, 2013; Gouveia et al., 2013], control and data dependency, bug reports [Le et al., 2014], and so on.

2.4.5 Filtering strategies

Information filtering is a method that copes with large amounts of information [Hanani et al., 2001]. It has been recently used in several domains to expose users to only relevant information. Filtering methods should consider the characteristics of the domain they will be applied. Also, a filtering approach considers the cognitive aspects that are related to the behavior of its users [Hanani et al., 2001].

In the SFL domain, filtering methods can be applied to reduce the amount of code to be inspected by developers, which may improve the debugging effectiveness. However, a filtering approach should be able to keep the faulty code among the reduced set of results.
2.4.6 SFL: a running example

To conclude this section, we show an example of how SFL works. Consider the `max` program (Figure 2.5), which returns the maximum element from an array of integers. There is a fault in line 4 (statement 2, block 1) of `max`, which is caused by an increment in the variable `i` before its use. If the greatest value is in the first position of the array, a failure will occur. Moreover, if the array has just one element, an exception will be thrown.

<table>
<thead>
<tr>
<th>Line</th>
<th>Statement</th>
<th>Block</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>int max(int[] array, int length)</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>1</td>
<td>{</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>int i = 0;</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>1</td>
<td>int max = array[++i]; // array[i++];</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>2</td>
<td>while(i &lt; length)</td>
</tr>
<tr>
<td>6</td>
<td>-</td>
<td>3</td>
<td>{</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>3</td>
<td>if(array[i] &gt; max)</td>
</tr>
<tr>
<td>8</td>
<td>5</td>
<td>4</td>
<td>max = array[i];</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>5</td>
<td>i++;</td>
</tr>
<tr>
<td>10</td>
<td>-</td>
<td>5</td>
<td>}</td>
</tr>
<tr>
<td>11</td>
<td>7</td>
<td>6</td>
<td>return max;</td>
</tr>
<tr>
<td>12</td>
<td>-</td>
<td>6</td>
<td>}</td>
</tr>
</tbody>
</table>

Figure 2.5: Code of the `max` program

Table 2.3 shows five test cases (t1, t2, t3, t4, and t5) of the `max` program. The test cases t4 and t5 fail, while t1, t2, and t3 pass. Table 2.4 presents the coverage matrix for statement and block spectra (indicated by the bullets associated with each statement, block, and test case) of `max`, the number of times whether an element is executed or not by failing and passing test cases, and the suspiciousness score assigned using the Ochiai metric.

<table>
<thead>
<tr>
<th>T_n</th>
<th>Test</th>
<th>Expected output</th>
<th>Actual output</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>max( [1,2,3] , 3 )</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>t2</td>
<td>max( [5,5,6] , 3 )</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>t3</td>
<td>max( [2,1,10] , 3 )</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>t4</td>
<td>max( [4,3,2] , 3 )</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>t5</td>
<td>max( [4] , 1 )</td>
<td>4</td>
<td>error</td>
</tr>
</tbody>
</table>

The suspiciousness score for block 1 is 0.63 and for the remaining blocks is 0.35 or 0. Therefore, the most suspicious elements are statements 1 and 2 according to Ochiai.
2.5 Final remarks

This chapter presented concepts related to debugging, testing, and automated debugging. We showed the main fault localization approaches proposed in the last decades, with special attention to spectrum-based fault localization, which is used in our research. In the next chapter, we present the related work.
Chapter 3

Survey and related work

This chapter presents the main advances and challenges in spectrum-based fault localization. In Section 3.1, we show the results of a survey that we conducted about SFL techniques. Section 3.2 presents related work. We conclude this chapter by presenting a concept map that shows where our contributions to the SFL area are.

3.1 Survey of spectrum-based fault localization

During this research, we conducted a survey [de Souza et al., 2017] to achieve a comprehensive overview of the state-of-the-art in SFL. In this section, we present our criteria to select relevant studies. Then, we discuss the main advances and challenges identified in these studies.

3.1.1 Selection criteria

We combined techniques of systematic review [Biolchini et al., 2005; Kitchenham, 2004] and backward snowballing [Jalali and Wohlin, 2012] to reach extensive coverage of the literature. We searched for studies published in the following digital libraries: ACM Digital Library, IEEE Xplore Digital Library, SpringerLink, and SciVerse Scopus - Elsevier. Other sources such as SciVerse ScienceDirect - Elsevier, Google Scholar, and DBLP Computer Science Bibliography were used, but no relevant additional studies were found. We focused on studies published from 2005 to October 2017, although we also included seminal studies published before this period. The studies included in this survey were published in journals and conferences with acknowledged quality.

We selected studies that: (1) proposed new SFL techniques; (2) carried out experimental evaluations of the proposed technique; (3) compared existing fault localization techniques; (4) proposed improvements to program spectra data (e.g., testing information); (5) assessed practical use of SFL techniques.

We read the title and abstract of all studies returned. In doubtful cases, other sections of the papers were read to decide whether they should be included. We read the selected papers in full. For the backward snowballing process, we first selected papers based on the description of such works from source papers. Then, we applied the criteria described above to select the most relevant studies.

We proposed a taxonomy to classify the papers regarding the research topics and challenges that characterize these studies. Figure 3.1 shows this taxonomy, which is composed of the following major topics: SFL techniques, Spectra, Faults, Programs, Testing data, and Practical use. Most of the studies intersect more than one topic. For each topic, we describe and discuss studies that present the most distinguishable contributions.

3.1.2 Advances and challenges

In this section, we present the main features, results, and challenges of fault localization techniques. We follow the structure proposed in Figure 3.1 to organize the discussion.
SFL techniques

Several ranking metrics have been proposed for fault localization, which were created or adapted from other areas. Each one has its specificity, as discussed in Section 2.4.1.

Experiments have shown that Ochiai has presented higher effectiveness compared to other ranking metrics [Abreu et al., 2009a; Le et al., 2013a; Xie et al., 2013]. However, the effectiveness of the best ranking metrics is slightly better (e.g., around 1% less code to examine) in most cases, indicating that they provide similar results [Debroy and Wong, 2011; Le et al., 2013a; Naish et al., 2011]. This means that there is a “ranking metric barrier” for such approach. Can we do better to distinguish suspiciousness values from the actual faulty elements? A possible way is to investigate whether ranking metrics present better results for different types of faults. For example, Sahoo et al. [2013] observed that Tarantula and Ochiai are very accurate for bugs that involve control-flow divergences. Indeed, almost all studies did not consider characteristics of faults to understand the limitations of their techniques. Combining previous SFL techniques to propose new ones requires deep knowledge of the strengths and weaknesses of each technique through different program characteristics, which can help to understand new ways to improve fault localization.

Other ways to enhance fault localization explore program behavior; program dependence and artificial intelligence techniques have been used for this purpose. Some of these techniques have presented promising results in precisely pinpointing faults. However, Artificial intelligence-based techniques often have high computational costs, especially for large programs. SFL techniques that use program dependence information often deal with large amounts of code. Thus, such approaches need an extra effort to develop strategies to reduce information. Such strategies may also impose high execution costs. Moreover, SFL techniques based on both artificial intelligence and program dependence can be used to identify relationships in the internal structures of programs, helping to provide contextual information about existing faults. Further studies shall be proposed to provide better results.

Spectra

The choice of program spectra influences the performance of SFL techniques, impacting on execution costs, data available to be analyzed, and outputs for inspection. Techniques that use coarser spectra data (e.g., class spectrum) may have reduced execution costs, requiring less code instrumentation.

Although statement spectrum is more often used, searching for faults through single statements may be difficult due to the absence of context that isolated statements provide for developers. To tackle this issue, grouping most suspicious statements from the same code regions can help to understand faults. Method spectrum may help to bring context to comprehend faults since it is the lowest code level that contains the logic of a program. Methods were also chosen as the preferred code unit for debugging by developers [Kochhar et al., 2016]. However, developers will manually inspect these methods, which may increase the amount of code to be verified compared to statement spectrum. An increasing number of techniques have used method spectrum [Laghari
However, these studies have not yet been compared among them. Class spectrum can also help in understanding faults, although classes often contain a significant amount of code.

SFL techniques can also provide new ways to explore program spectra data. For example, predicate spectrum provides information about the executed entities and their results (true or false) [Liblit et al., 2003; Naish et al., 2010]. Thus, besides the information about whether a statement is executed by a test case, it is also possible to verify its boolean result to improve SFL.

Other studies combine different spectra [Le et al., 2016; Masri, 2010; Santelices et al., 2009; Yu et al., 2011]. We note that data-flow information can improve fault localization techniques. However, the amount of collected data and the execution costs to process it are high compared to those based on control-flow. Maybe due to this fact, little research has explored the use of data-flow for SFL. Strategies to reduce execution costs to collect data-flow spectrum can make these approaches more feasible for practical use [Chaim and de Araujo, 2013; de Araujo and Chaim, 2014]. We discuss more data-flow spectrum in Section 3.2.6.

Faults

Most of the proposed SFL techniques have been assessed using single-fault programs. From those that are assessed using both single and multiple faults, the multiple-fault assessments are often treated as secondary studies. These studies carried out complete experiments for programs with single faults, and small experiments with multiple-fault programs [Wong et al., 2012; Zhang et al., 2009]. In some cases, experiments with multiple faults use a reduced number of programs when compared to single-fault experiments.

SFL techniques have been proposed to deal with multiple faults [Dean et al., 2009; Jones et al., 2007; Xue and Namin, 2013]. However, their performances have not been compared among these studies. The absence of programs containing multiple faults makes it difficult to conduct experiments. Often, studies that perform experiments with multiple faults create their multiple-fault versions by randomly merging single-fault versions [Debroy and Wong, 2009; Jones et al., 2007; Naish et al., 2009]. This fact impairs the comparison between techniques, due to the different procedures used to create the benchmarks.

The presence of multiple faults has been shown to hamper fault localization effectiveness [Naish et al., 2009] (see Section 2.4.2). SFL techniques that cope with multiple faults often add more complexity to the fault localization process to isolate such faults. On the other hand, DiGiussepe and Jones [2015] argued that multiple faults had a negligible impact on effectiveness since at least one of the faults is well ranked in most cases. Their results show that SFL techniques had an average 2% decrease in effectiveness. For large programs, though, 2% of statements may represent a sizable amount of code for inspection. The existence of multiple faults can even increase fault localization effectiveness. When two or more faults cause failures on different test cases, it is possible that at least one of these faults is well ranked, as occurred in our study [de Souza et al., 2018] (see Chapter 4). The study of Perez et al. [2017] investigated bug fixing commits from 72 open source projects, showing that 82.5% of the commits have a single fault. This fact may alleviate the need for more complex techniques to deal with multiple faults. However, unknown faults may exist in such commits, which may impact on the fault localization results.

Another important issue raised by recent studies is the interference between simultaneous faults [Debroy and Wong, 2009; DiGiussepe and Jones, 2015]. More studies are necessary to investigate the effects of fault interference on fault localization. Existing studies have already shown that the interference among faults is very frequent. One concern is that experiments with faults randomly spread across programs do not assure that the faults indeed interfere with each other, which may result in non-realistic multiple-fault programs.

The behavior of SFL techniques in the presence of different fault types is a rarely approached issue. Authors have reported their techniques’ difficulties in dealing with particular fault types. Some techniques explicitly do not identify faults resulting from code omission [Masri, 2010; Yu et al., 2016; Sohn and Yoo, 2017; Zhang et al., 2017].
et al., 2011]. A few works analyzed the behavior of their techniques for specific faults [DiGiuseppe and Jones, 2012; Liblit et al., 2005; Mariani et al., 2011]. By assessing the performance of SFL techniques through different fault types, it is possible to understand strengths and weaknesses of such techniques and, thus, propose new ones that are more effective to cope with specific fault types. It is also possible to improve a technique that does not work well for a fault type or even combine techniques that are better for distinct fault types.

### Programs

The use of small programs facilitates experimentation. It also allows different studies to use the same subject programs to compare techniques. However, the prevalence of small programs and the use of the same benchmark programs in experiments impairs the assessment of SFL techniques. Thus, the use of programs from different domains is needed for a comprehensive evaluation of SFL techniques.

As discussed in the previous subsection, programs used in SFL studies do not contain multiple simultaneous faults. Thus, the creation of programs with multiple faults can contribute to experimentation in more realistic scenarios. Creating benchmarks for controlled experimentation is expensive and difficult [Do et al., 2005]. One way to obtain new benchmarks is to identify faults in software repositories of open source programs, as done by Dalmeier and Zimmermann [2007] and Just et al. [2014]. This approach provides real faults for experimentation and facilitates reproduction of experiments. Another possible solution to reduce the effort in creating benchmarks is to seed faults by using mutation testing, which has been shown to be a good simulation of real faults [Ali et al., 2009]. Conversely, Pearson et al. [2017] showed that SFL techniques present different results for seeded and real faults.

Notwithstanding the current existing benchmarks, new programs should be added to increase the diversity of domains and faults for experimentation. For example, most of the real programs currently used to evaluate SFL techniques are frameworks and libraries. End-user programs should also be assessed to comprehend the performance of SFL better. These new programs must also be available to other researchers to facilitate their use in future studies.

### Testing data

Software testing is the main source of information for debugging. There are several ways to measure test quality. Testing requirements are used to guarantee that the code is widely tested, and most of the program elements are executed. Fault detection rate is a quality measure used to assess the ability of a test suite to reveal failures. Fault localization leads to another desired criterion for test suites: fault localization capability. This characteristic means that test suites should be able to distinguish program elements from their test cases [Baudry et al., 2006].

A natural process for obtaining test suites with higher coverage is to increase their size. However, large test suites lead to greater computational costs to execute them. Test suite reduction strategies are then used to minimize the number of test cases without losing the ability to detect failures [Yu et al., 2008; Zhang et al., 2015]. Moreover, test suites are also expected to hold the distinctiveness of program elements throughout the test cases, i.e., the fault localization capability. Thus, test suite reduction techniques have to cope with a trade-off between reducing the test size and keeping test cases distinguishable.

Future studies should devise new ways to measure the quality of testing data for fault localization. Test cases that cover a reduced amount of code excerpts may be useful to highlight the most suspicious ones. Conversely, individual test cases that cover large code excerpts may impair fault localization by adding an excessive amount of program entities for each execution. Thus, such studies may provide ways to identify needs for improvements and the suitability of test cases for use in fault localization.
Practical use

Evaluation metrics used to measure SFL techniques are based on assumptions that in practice may not occur. Measuring fault localization performance by the relative position of a faulty entity in a ranking list can mislead the effort to find bugs. For example, if a technique returns the faulty entity within 1% of the most suspicious statements of a program with 100 KLOC, it may be necessary to inspect 1 KLOC to find the fault. This may be infeasible in practice. Developers tend to leave SFL lists if they do not find the fault among the first picks [Parnin and Orso, 2011]. As pointed out by previous studies [Parnin and Orso, 2011; Steimann et al., 2013], the techniques should focus on the absolute position, with the faulty statement among the first positions. Indeed, some recent studies have focused on assessing their techniques using absolute ranking position [de Souza et al., 2018; Le et al., 2015a; Sohn and Yoo, 2017; Steimann and Bertschler, 2009].

The ranking lists provided by SFL techniques are based on suspiciousness scores and may be composed of sequences of entities with no direct relation to each other (e.g., statements that belong to distinct classes). This fact may impair the identification of faults. Thus, techniques that provide more information can help developers to understand the conditions in which faults occur. Strategies have been proposed to tackle this problem, such as grouping related entities, exploring different code levels, adding source code information, or presenting subsequences of execution traces. Future approaches should explore new ways to reduce the amount of non-relevant information and to enrich the information provided to developers.

When developing fault localization techniques, we suppose several assumptions about the developers’ behavior while performing debugging tasks. However, without user studies, one cannot know whether these assumptions hold in practice. Thus, these studies are essential for identifying how the techniques are used, to assess the developers’ willingness to use SFL techniques, and to provide guidance on their use in industrial settings. Unfortunately, debugging user studies are still scarce and the few existing ones have presented divergent results. This may occur due to difficulties to obtain participants for user studies, especially professional developers, and samples that are statistically representative. Notwithstanding, new user studies are pivotal for understanding the feasibility of SFL adoption by practitioners.

Moreover, there is a lack of evaluations of SFL techniques in real development scenarios (e.g., case studies), which can bring findings to improve fault localization. Also, the current user studies had participants without previous knowledge of the code being debugged. Can SFL improve the debugging performance of developers inspecting their code? Future studies should consider these concerns aiming to understand the use of SFL in practice better.

3.2 Related work

We presented the related work according to the different aspects we addressed in our research. First, we present works related to spectrum-based fault localization. Then, we present works that proposed contextualization for SFL. Next, we show user studies on SFL and works that used the data-flow spectrum for SFL. We conclude this section with a concept map that clarifies our contributions to the SFL area.

3.2.1 Spectrum-based fault localization

Jones et al. [2002] presented a technique called Tarantula, which uses a homonym ranking metric to classify suspicious statements. These statements are shown in a graphic form of visualization, using different colors according to their suspiciousness values. Abreu et al. [2009a] proposed the use of Ochiai and Jaccard similarity coefficients as fault localization ranking metrics. The results show that both Ochiai and Jaccard outperform Tarantula on fault localization effectiveness.

Several other ranking metrics were proposed for fault localization, which led researchers to compare their performance. Xie et al. [2013] performed a theoretical comparison between 30 ranking metrics used in fault localization. To make this comparison, they grouped statements classified by
the ranking metrics according to statements with respective higher, equal, and lower scores than the faulty statement. They identified six groups of equivalent ranking metrics and other seven metrics that lack equivalent ones. They also showed that five ranking metrics (O, O³, Wong1, Russel & Rao, and Binary) perform hierarchically better than others for two single-fault sample programs. Le et al. [2013a] evaluated the study of Xie et al. [2013] using well-studied programs, showing that the theoretical comparison does not hold for such programs.

Ranking metrics have also presented divergent results regarding programs with real and seeded faults. The study of Pearson et al. [2017] shows that the Metallaxis [Papadakis and Le Traon, 2015] and O³ ranking metrics perform better for seeded faults, while DStar and Ochiai performed better for real faults. Thus, several factors should be considered when using ranking metrics, since their performance may vary on different study settings.

The works presented above show that the fault localization performance may vary across programs and metrics. Thus, it is important to use metrics that present divergent results and programs with different characteristics to assess whether the performance of a technique is independent of the experimental setup.

### 3.2.2 Contextual information

Some recent work have addressed the use of contextual information for fault localization. Jiang and Su [2007] proposed a technique based on Machine Learning to find predicates that correspond to failures from control-flow paths. Developers can inspect these paths to search for faults. The technique presented by Hsu et al. [2008] analyzes parts of failing execution traces to identify subsequences of statements frequently executed, which can be examined to search for faults. These subsequences are called bug signatures; each of them may contain one or more branches in their execution order. Cheng et al. [2009] proposed a model of subgraphs composed of nodes (methods or blocks) and edges (method calls or method returns). These subgraphs are obtained from the difference between failing and passing control-flow executions. A graph mining algorithm is then applied to obtain a list of the most suspicious subgraphs.

Liu et al. [2010] proposed a technique that uses variance analysis to locate faults in program behavior patterns. These patterns are obtained from the behavior of objects that belong to the same class. The technique indicates the suspicious pattern messages that occur in a class during a failing execution. DiGiuseppe and Jones [2012] presented the Semantic Fault Diagnosis (SFD) technique. SFD gathers information from the source code, such as developer’s comments, class and method names, and variable expressions. The technique instruments a program and parses its source code to obtain the needed information. Terms of the source code are normalized and correlated with the program entities they belong to. The outcome is a list of top terms related to a fault.

Other studies use information from the structure of programs to improve fault localization. Gouveia et al. [2013] proposed a technique that provides contextual information by grouping statements according to their code structure. Their technique assigns, for each code level (e.g., packages, classes, methods), the highest suspiciousness of its internal statements. The work of Li et al. [2014] considers that some structures can influence their statements, making them more susceptible to SFL techniques wrongly classifying them. For example, a faulty statement in the main class can be underestimated because it is always executed by both failing and passing runs. Conversely, a non-faulty statement in a catch block can be overestimated because it is only executed by failing runs.

Le et al. [2015b] proposed a technique—called Adaptive Multi-modal bug Localization (AML)—that combines spectrum- and information retrieval-based fault localization. AML uses program spectra, text from bug reports, and suspicious words—words from bug reports that are also present in source code comments—to indicate methods that are more likely to be faulty. Youm et al. [2017] use information from bug reports, stack traces, source code structure, and source code changes in their information retrieval (IR)-based bug localization technique—called Bug Localization using Integration Analysis (BLIA). BLIA provides file- and method-level information for inspection.
3.2.3 Method-level spectrum

The search for bugs in methods or functions has been a goal since the very inception of automated debugging. Shapiro [1983] proposed an algorithm to automatically locate the faulty procedure in a program. His approach requires that the developer answer whether the outputs of procedure calls are correct for a particular set of parameters. Unfortunately, such an approach does not scale for large programs, since many procedure calls should be checked.

Dallmeier et al. [2005] used method call sequences occurring in test executions to indicate classes that are more likely to contain faults. Burger and Zeller [2011] used method call coverage to obtain information from interactions between methods that occur during the execution of a failing test case. After applying delta debugging [Zeller, 2009] and program slicing, this information was used to generate a minimal set of interactions relevant to the failure. Mariani et al. [2011] presented a technique—called Behavior Capture and Test (BCT)—that uses method coverage to generate an interaction model. Interactions are sequences of method calls. This model is compared with interactions from failing test cases to indicate method calls that are more likely to contain faults. Shu et al. [2013] presented a technique called Method-Level Fault Localization (MFL). MFL uses test execution to construct a dynamic call graph and a dynamic data dependence graph of inter-method data. These graphs are merged and applied to a causal inference technique to create a list of suspicious methods.

Other recent studies have also addressed the use of method-level information for fault localization. Kochhar et al. [2016] conducted a survey study with 386 professional practitioners, asking for their expectations about fault localization research. Regarding the granularity level provided by fault localization techniques, most respondents prefer to inspect methods, closely followed by statements and blocks. Classes and component were less preferred. Laghari et al. [2016] proposed a technique—called pattern spectrum analysis—that provides a list of suspicious methods. They identify patterns of sequences of method calls during the execution of integration testing. A frequent itemset mining algorithm is used to select patterns for each method within a minimum support threshold. Each method is assigned with the highest score from its respective patterns of method calls.

Le et al. [2016] also presented a method-level fault localization technique, called Savant. First, they cluster methods executed by failing and passing test cases. They generate method invariants for the clusters created in the previous step. Invariants that differ between passing and failing executions are deemed as suspicious. Then, they apply ranking metrics and use these results along with the invariants as an input for a learning-to-rank machine learning technique to produce a ranking model. This model is used to create lists of suspicious methods for faulty programs. Musco et al. [2016] presented a technique, called Vautrin, that provides a list of suspicious methods. They use mutants to identify call graph edges (i.e., method calls) more likely to contain faults. The rationale is that if a mutant is killed by a test, some of its edges can be related to a fault.

3.2.4 Absolute ranking

Recent fault localization techniques were assessed using the absolute number of programs elements inspected until reaching a fault [Laghari et al., 2016; Le et al., 2015b, 2016; Li et al., 2016; Musco et al., 2016]. The use of absolute positions seems to be more suitable to real debugging scenarios than the percentage of inspected code [Parnin and Orso, 2011; Steimann et al., 2013]. In the study of Kochhar et al. [2016], a large number of respondents considered using a fault localization technique that leads them to inspect at most five program elements.

Some studies evaluate fault localization techniques by restricting the number of program elements to be inspected. They use absolute values, i.e., effort budgets, to measure their fault localization effectiveness. Le et al. [2014] proposed a technique to evaluate the output of SFL techniques to indicate whether this output should be used by developers. To evaluate the technique, they used the top ten positions of an output to verify when the approach is effective for classifying such outputs.
3.2.5 User studies

From the automated debugging techniques proposed in the last decades, Spectrum-based fault localization (SFL) techniques are promising due to their relative lightweight execution costs and their intuitive assumption of suggesting elements more likely to be faulty. To further advance the techniques for practical use, experiments with developers are needed to evaluate theoretical assumptions of SFL and to assess whether and how they use SFL. Below, we present user studies that addressed debugging, automated techniques, and SFL techniques.

User studies on debugging and fault localization

Several debugging techniques have been proposed since the late 1950s [Gilmore, 1957]. However, few user studies are assessing these techniques [Parnin and Orso, 2011]. Most of these assessed the practical use of program slicing techniques [Francel and Rugaber, 2001; Ko and Myers, 2009; Kusumoto et al., 2002; Weiser, 1981, 1982]. Francel and Rugaber [2001] noted that only 3 of 111 papers on slicing-based debugging techniques had addressed practical use issues by that date. Another user study assessed the human accuracy at locating faults [Fry and Weimer, 2011]. Youngs [1974] evaluated faults from 42 developers to identify what faults they made and how their experience affects the occurrence of these faults. Gould [1975] showed in his user study that developers often use several information sources during debugging, from variable names to source comments. The author claimed that developers frequently search for clues in the source code while debugging.

Other studies have been proposed to understand the developers’ behavior during debugging. Vessey [1985] suggests that experienced developers can chunk code while debugging. They use breadth-first strategies associated with a system view of the domain. Novice developers are less able to chunk code and use depth-first strategies. They also use breadth-first approaches, but with less ability to think in terms of a system domain. Gilmore [1991] stated that comprehension process is completed before debugging occurs, but comprehension and debugging are strongly related. He also claimed that debugging models proposed until then were simplistic by considering comprehension and debugging separately. By comprehending a program, developers should have their debugging tasks facilitated.

Other user studies regarding debugging tools were proposed. Ko and Myers [2009] evaluated their Whyline debugging tool1 [Ko and Myers, 2008] with 20 developers. Whyline provides questions to assist developers in understanding a program’s behavior. The tool uses static and dynamic slicing to formulate questions, presenting such questions graphically and interactively. Two real faults from ArgoUML (with 150 KLOC) were used in the experiments. Using Whyline, more developers located the bugs compared to those that did not use the tool. Developers were also faster using the tool. Wang et al. [2015] evaluated whether fault localization techniques based on information retrieval (IRFL) help developers to locate faults. The study had 58 participants who performed two debugging tasks, interchangeably using and not using the IRFL tool. The authors selected eight bugs from the SWT project2. The results show that bug reports with more useful information (i.e., descriptions of program entities) helped developers to identify the faulty files faster, but did not improve in locating the bugs. Good IRFL suspiciousness lists (when a faulty file is ranked first) helped developers more when the respective bug report had no information about program entities.

User studies on spectrum-based fault localization

Parnin and Orso [2011] proposed the first study to assess the practical use of SFL. They evaluated several theoretical assumptions that SFL studies make about how developers use SFL. 34 participants took part in two experiments, performing two debugging tasks in Tetris3 (with 2.4 KLOC) and NanoXML (with 4.4 KLOC) [Do et al., 2005]. In the first experiment, 24 participants

---

1github.com/andyjko/whyline
2eclipse.org/swt
3percederberg.net/games/tetris
used SFL in one of the tasks, and in the second experiment, the ten other ones used SFL in both
tasks. The authors found that SFL improved the performance of more experienced developers.
Moreover, developers did not follow the suspicious lists in order while debugging and did not find a
bug immediately after inspecting it. They also suggest that SFL techniques should focus on ranking
faulty statements among the first positions to avoid developers giving up on using such tools.

Gouveia et al. [2013] performed a user study to evaluate the GZoltar tool\[Campos et al., 2012\].
GZoltar is an Eclipse plug-in that provides three different visualizations to report SFL results. These
visualizations allow developers to investigate the structure of OO Java projects, showing the most
suspicious, packages, classes, methods, and statements. The study was performed by 40 participants,
who searched for a fault in XStream\[X-stream.github.io/repository.html\] (with 17.4 KLOC). The 20 participants that used GZoltar
found the bug, while only 7 participants found without using it. The developers were also faster
to locate the fault using GZoltar. The tool received positive evaluations regarding its usability.
Gouveia et al. [2013] study is more focused on the developer’s performance using the tool, and not
in the behavior of programmers while debugging.

Xie et al. [2016] conducted a study with 207 participants and 17 debugging tasks. Each partici-

cipant performed two tasks, with and without using an SFL tool. Seven standard Computer Science
algorithms (e.g., quicksort, heapsort), with at most 500 LOC, were used in the experiments. Four
of these programs contain a single file. The faults were seeded by the authors and categorized into
six groups. Their results show that SFL helped the participants to locate faults when such faults
were well ranked (i.e., ranked among the most suspicious statements). The results also show that
SFL can decrease debugging efficiency, even for the well-ranked faults. Moreover, the authors found
that most participants started with a first scan (i.e., a brief overview) of the code before using the
SFL lists.

Xia et al. [2016] presented a user study about SFL with 36 professional developers. They used
16 real bugs from four open source projects. The participants were divided into three groups: using
an accurate SFL tool, a mediocre SFL tool, and without using SFL. The accurate tool ranks the
faults among the top-5 positions, while the mediocre tool ranks them between top-6 and top-10.
Each participant performed four debugging tasks. Their results showed that developers using the
accurate tool were more effective and efficient than those using both the mediocre one and without
using SFL. The developers using the mediocre SFL tool were also more effective than those without
using SFL.

3.2.6 Data-flow spectrum

Despite the ability to exercise different aspects during test execution, data-flow is rarely used to
support SFL. Few initiatives use definition-use associations (duas) [Masri, 2010; Santelices et al.,
2009] while others use program slicing [Ju et al., 2014; Lei et al., 2012] or different types of data
and control dependencies [Eichinger et al., 2010; Masri, 2010; Yu et al., 2011]. All these approaches
demand a costly time overhead to collect data-flow spectrum.

Santelices et al. [2009] proposed a technique, avg-SBD, that combines different spectrum types
for fault localization: statements, branches, and du-pairs—a variant of data-flow information that
takes into account only c-use duas. They also introduce an approximate dua spectrum, avg-SBD
approx, which has lower overhead at run-time. The results, assessed using small- to medium-sized
programs, showed that their approach locates more faults than each spectrum isolated. They also
showed that different faults are best located by different coverage types.

Masri [2010] uses a spectrum called Dynamic Information Flow Analysis (DIFA) for fault lo-

calization. DIFA is composed of interactions performed in an execution, including inter-procedural
data- and control-flow dependences from statements and variables. The author compared DIFA with
statement, branch, and dua spectra using 18 faults of medium-sized programs. DIFA presented bet-
ter performance to locate the bugs. However, the time to generate DIFA spectrum was \(2\text{–}10\) times

\footnote{gzoltar.com}
\footnote{x-stream.github.io/repository.html}
slower than dua and branch spectra. In this study, executing DIFA on the NanoXML program (7.6 KLOC) took 64 minutes and needed 377 MB of disk space. For the JAligner program (3.2 KLOC), DIFA took 260 minutes and needed 384 MB of disk space.

Hemmati [2015] assessed the ability of data- and control-flow coverage criteria to reveal faults on programs from the Defects4J database. Du-pair criterion covered 79% of faults that were not covered by control-flow criteria. He also showed that 15% of all faults were not covered by any criterion. The author did not report issues regarding run-time costs. Yu et al. [2011] presented the LOUPE technique, which combines data- and control-dependencies for fault localization. The output is a list of statements. Their experiments using small programs from the Siemens suite [Hutchins et al., 1994] presented better performance than the compared techniques. They also assessed LOUPE in the space program [Do et al., 2005] (6.2 KLOC), which took 100 minutes to execute. Eichinger et al. [2010] propose the data-flow enabled call graphs (DEC graphs), which is a method call graph with data-flow information. DEC graphs also register parameters and method-return values. To reduce the number of possible parameter values, they discretize numerical parameter and return values using Data Mining approaches. They assessed the technique including ten faults in the Weka program (301 KLOC). The authors did not report the technique’s run-time overhead.

3.2.7 Comparison with previous work

In our study of contextual information (see Chapter 4), we use information of code level structures as in Gouveia et al. [2013] and Li et al. [2014]. One of our techniques, called Code Hierarchy (CH), uses an approach similar to GZoltar to compute the suspiciousness of methods: we assign to a method the highest suspiciousness score of its internal blocks. Different from GZoltar, CH establishes a list of methods for inspection and uses the number of blocks with the maximum suspiciousness score as a tie-breaker for methods with the same score. Our other technique, Integration Coverage-based Debugging (ICD) differs from previous work by ranking pairs of method calls (caller-callee methods) executed by the test run. Moreover, CH and ICD use filtering strategies to reduce the amount of code to inspect. Beyond the use of code level structure for contextualization, data-flow spectrum (see Chapter 6) may be an alternative way to contextualize fault localization, since it provides information about suspicious variables.

The Jaguar fault localization tool, used in our user study (see Chapter 5) has views that provide information of different code level structures (packages, classes, method), as in the GZoltar tool. Jaguar also colors suspicious code, as done by Tarantula and GZoltar, to facilitate code inspection. Additionally, Jaguar provides information of data-flow spectrum, more specifically definition-use associations (duas). Differently from GZoltar, the Jaguar views are presented as lists of suspicious code, which are straightforward as the other Eclipse’s views, such as the JUnit and the Package Explorer views. However, the Jaguar’s views that contain data-flow, classes and packages were not used in our experiments to avoid the presence of new independent variables in our analysis.

Differently from other techniques that provide method-level information, our fault localization techniques of Chapter 4 also include block-level information for code inspection. Thus, two code levels are available for fault localization. The Jaguar tool also provides a list of methods for participants of our user study.

User studies have presented divergent results regarding the usefulness of SFL in practice, which encouraged us to further investigate the factors that can lead developers to accept or refuse the use of SFL techniques. As in previous work, we investigate both effectiveness and efficiency of SFL compared to traditional debugging. We also analyze issues regarding the developers’ behavior when using SFL and whether their programming experience affects their performance using SFL.

Different from the previous user studies, we investigate if the granularity of suspicious information impacts on the developers’ performance. We use two levels of suspicious information: lines and methods. Moreover, we investigate whether SFL can be useful for guiding developers to inspect the

---

6 github.com/rjust/defects4j
faulty code. For this purpose, we evaluate if SFL leads developers to identify the faulty method and to inspect the faulty line more than traditional debugging. Another difference of our study is that we evaluate the developers’ intention to use SFL in the future, which can help to understand if developers are actually willing to use SFL tools.

Our study about data-flow spectrum compares data- and control-flow spectra to investigate their effectiveness for fault localization, as previously done by Santelices et al. [2009] and Masri [2010]. Differently from the previous works, we assessed both spectra using ten different ranking metrics. Both Santelices et al. [2009] and Masri [2010] used Tarantula in their evaluations, which is overcome by other metrics in recent studies [Le et al., 2013a; Xie et al., 2013]. Moreover, we used medium- to large-sized programs containing real faults. We also compared the overhead of SFL techniques for both spectra, showing that data-flow can be used at an affordable cost. Finally, we carried out statistical analysis in our assessment, showing that data-flow spectrum improves SFL with statistical significance.

Regarding the evaluation metrics, our studies in Chapters 4 and 6 use the absolute ranking as a criterion for assessing the fault localization effectiveness. By doing this, we evaluate the techniques using values that are feasible for use in realistic scenarios.

3.3 Overview of the contribution of this thesis

Based on the analysis process carried out in our survey, we created a concept map [Novak and Cañas, 2008] that presents an overview of the main characteristics derived from SFL studies and their relationships within the area, which is shown in Figure 3.2. We use this concept map to highlight where are our contributions and what are the concepts addressed in this thesis.

Figure 3.2 presents the multiple challenges that should be tackled to improve SFL for its main objective: using SFL for debugging in real development settings. Below, we briefly describe our contributions to each topic. These contributions, highlighted in blue, are related to the concepts shown in Figure 3.2.

We proposed new SFL techniques that provide contextual information for fault localization. We used different program spectra in our studies, which includes method and data-flow. We proposed ranking lists composed of two levels of information: methods and their respective blocks. We carried out an experiment with multiple faults to evaluated how our contextualization techniques perform on such program subjects.

Our experiments counted with real and large open source programs. We evaluated our studies using the absolute position of a fault in the SFL lists as the evaluation metric criterion. Our study comparing data- and control-flow spectra was assessed using ten ranking metrics. We also performed a user study to assess how developers use SFL in practice.

All of our experiments’ decisions were guided by the search for new ways to improve SFL for its practical use, aiming at collaborating to the adoption of automated debugging in real development settings.

3.4 Final remarks

In this chapter, we presented several studies that are related to our research. Also, we presented the results of a survey of spectrum-based fault localization that we performed, showing the main challenges and advances in the SFL area. We concluded by showing where our contributions fit in the SFL area using a concept map that we proposed as a result of the SFL survey. The following chapters describe the studies we carried out during this research project, including their methodology, results, and findings.
Figure 3.2: Spectrum-based fault localization concept map
Chapter 4

Adding contextual information for spectrum-based fault localization

In this study, we propose new ways to contextualize spectrum-based fault localization. Most SFL techniques provide lists of suspicious statements, which may contain statements from different code excerpts among the first picks. Other techniques indicate the most suspicious methods.

In this chapter, we present and discuss our proposal to improve SFL by combining block- and method-level information. We use two techniques called Code Hierarchy (CH) and Integration Coverage-based Debugging (ICD) to create lists of suspicious methods and their respective blocks. We also propose two new filtering strategies called Level Score (LS) and Fixed Budget (FB) to reduce the number of blocks to inspect in each method.

We present Road2Fault, a tool that implements our techniques and filtering strategies. Then, we describe the methodology, results, discussion and threats to validity. Finally, we draw our final remarks regarding this study.

4.1 Research questions

We retake our research questions related to this study, as previously presented in Section 1.2. The main question that guides this study is: Can contextual information improve SFL effectiveness? The following specific questions aim to answer our concerns regarding this study.

RQ1: Are roadmaps combined with filtering strategies effective for fault localization?

RQ2: Which is the best level \( (l) \) value to use in the Level Score filtering strategy?

RQ3: Which is the most helpful filtering strategy for fault localization?

RQ4: How many methods are inspected to reach the faults using roadmaps and filtering strategies?

RQ5: Are roadmaps combined with filtering strategies effective for programs containing multiple faults?

4.2 Contextualizing fault localization

In this section, we present techniques to support contextual guidance for fault localization. The contextualization strategy proposed is two-fold. It comprises the selection of most suspicious methods and also the selection of blocks that should be inspected at each method. Thus, the techniques aim to reduce the amount of code to be inspected using method-level information. Moreover, grouping blocks by their methods can add more information to understand faults.
Code Hierarchy (CH) and Integration Coverage-based Debugging (ICD) provide method-level information through a roadmap for fault localization. A developer searching for a bug investigates the most suspicious methods and, for each method, investigates its most suspicious blocks. Both CH and ICD were proposed in my Master’s thesis [de Souza, 2012].

We then introduce two filtering strategies to narrow down the number of blocks to be inspected. The strategies are called Level Score (LS) and Fixed Budget (FB). Both strategies aim to reduce the effort needed to search for bugs inside of the methods. We show an example in which CH and ICD roadmaps are used to locate faults along with LS and FB filtering strategies.

4.2.1 Code hierarchy

Code Hierarchy (CH) [de Souza, 2012] is a technique based on the structure of object-oriented programs. It uses the hierarchical code structure of Java (packages, classes, and methods) to organize the debugging information.

CH assigns suspiciousness scores to packages, classes, and methods based on the suspiciousness scores of their internal blocks. The CH suspiciousness score is given by a pair \((susp, number)\) where \(susp\) is the highest suspiciousness score assigned to a block belonging to the program entity (package, class, or method) and \(number\) is the number of blocks that have such a score. Only block spectrum is required to compute CH.

In the case of a tie, the number of blocks with the same \(susp\) score (\(number\)) determines which entity will be investigated next. Ties in ranking lists are common occurrences that impair the performance of SFL techniques [Wong et al., 2010; Xie et al., 2010]. Xie et al. [2010] present four tie-breaking strategies for ranking lists, while Wong et al. [2010] assessed ties as an impact factor regarding the performance of fault localization techniques.

CH generates a roadmap—a list of suspicious methods—to be visited during fault localization derived from CH. A CH’s roadmap suggests an order for fault examination by establishing which methods should be inspected first. The rationale is that a method is a self-contained contextualized fault localization unit—it is the lowest code level structure encapsulating blocks associated with the logic of a program’s function. A developer starts by inspecting the most suspicious method of the roadmap and then investigates the method’s most suspicious blocks. If the developer does not locate the fault in the first method, s/he investigates the second one. This process continues until locating the bug or giving up on using the roadmap.

The motivation for using the CH roadmap and block spectrum lies in the assumption that the suspiciousness of a faulty block and its method are correlated. If a block is often executed by failing test cases, it will likely receive a high score, which leads to a well-ranked method. As a result, the faulty method tends to be well ranked in the roadmap because it contains the faulty block that causes a test to fail.

Algorithm 1 shows how the CH roadmap is determined. The input consists of a list of suspicious blocks and a list of methods. The suspiciousness scores are assigned to blocks according to a ranking metric \(M\), such as Ochiai or Tarantula. To create the CH roadmap, each method is assigned with the highest suspiciousness score of its blocks (\(susp\)) and the number of blocks with that score (\(number\)). Next, the methods are sorted by their \(susp\) score in descending order. CH establishes that the \(susp\) score is the most important factor for fault localization. For tie cases, we assumed that the smaller the \(number\), the better. Thus, if a method has only one block with the highest suspiciousness score, it should be investigated first. The idea behind first inspecting methods with the fewest most suspicious blocks is that these methods contain blocks that are better discriminated by ranking metrics, which in turn may lead to the faulty block. Moreover, it may be less tiresome for developers to inspect fewer blocks first if the bug is not in the method. In case of a double tie, the next entity is randomly decided.
4.2 CONTEXTUALIZING FAULT LOCALIZATION

4.2.2 Integration coverage-based debugging

Integration Coverage-based Debugging (ICD) [de Souza and Chaim, 2013] is another technique for tackling the lack of context in SFL techniques. ICD also provides a roadmap to investigate the most suspicious methods.

To create the roadmap, ICD uses a spectrum called MethodCallPair (MCP) [de Souza, 2012]. The ICD roadmap is created from a ranking of pairs of method calls—caller-callee relationships—performed during test execution. At each method, block spectrum is also used to locate bugs. Thus, ICD combines MCP spectrum to create the roadmap with block spectrum to fine tune the inspection. In the following subsections, we present the MCP spectrum and how the ICD roadmap is built.

MethodCallPair

The MethodCallPair (MCP) spectrum was developed to collect information from the most suspicious method call sequences. MCP was inspired by integration testing techniques [Lemos et al., 2009], but it was not devised as a testing requirement. Rather, it was conceived as integration information captured during a test suite execution for debugging purposes.

MCP represents relationships between methods executed during test runs. A pair is composed of a caller method and a callee method. Instead of simply capturing the method call stack trace to calculate the suspiciousness of methods, the idea is to highlight methods more often related to other methods of failing executions. Thus, the integration between methods is used to indicate those more likely to contain faults. Figure 4.1 illustrates MCP spectrum information, in which the method caller of class A invokes the method callee of class B.

Creating the ICD roadmap

The ICD roadmap is a simplified guide for inspecting a list of suspicious method call pairs (MCPs) to be used when searching for faults. ICD creates the roadmap from the MCP spectrum
Figure 4.1: MethodCallPair (MCP)

According to the following steps:

1. MCPs are tracked during the execution of each test case.
2. ICD uses the MCP spectrum to assign suspiciousness scores to each MCP. Any ranking metric $M$ can be used for this purpose.
3. ICD generates a list of MCPs sorted by suspiciousness. MCPs with the same suspiciousness scores are sorted according to their order of occurrence in the execution.
4. ICD creates the roadmap by visiting the MCP list from the top-ranked elements to the least ranked ones, and from the caller method to the callee method. If a method does not yet appear in the roadmap, it is added to the roadmap with the same suspiciousness score assigned to the visited pair.

Algorithm 2 shows how the ICD roadmap is created.

**Input:** mcpSortedList – an MCP list sorted in decreasing order of suspiciousness assigned by a metric $M$.

**Output:** icdRoadmap – an ICD roadmap.

```
icdRoadmap ← ∅;
while not mcpSortedList.empty() do
    mcp ← mcpSortedList.removeFirstElement();
    if not mcp.caller in icdRoadmap then
        method ← mcp.caller;
        method.susp ← mcp.susp;
        icdRoadmap.addLast(method);
    end
    if not mcpcallee in icdRoadmap then
        method ← mcp.callee;
        method.susp ← mcp.susp;
        icdRoadmap.addLast(method);
    end
end
return icdRoadmap;
```

Algorithm 2: ICD roadmap creation

Consider that an MCP (A.caller, B.callee) is executed in 4 out of 10 failing test cases, and in only 2 out of 90 passing test cases. Thus, the values for the $c_{ef}$, $c_{nf}$, $c_{ep}$, and $c_{np}$ spectrum coefficients will be 4, 6, 2, and 88, respectively. Using the Ochiai metric (see Table 2.2), the suspiciousness score will be 0.51 for the MCP (A.caller, B.callee). Once all MCPs have been assigned their scores, they are sorted in decreasing order; the list of sorted MCPs is then used to create the roadmap.
4.2 CONTEXTUALIZING FAULT LOCALIZATION

roadmap creation, if A.caller is yet not in the roadmap, it is inserted with a suspiciousness score of 0.51. Next, B.callee is checked. If B.callee is not yet in the roadmap, it is also inserted with a suspiciousness score of 0.51.

Therefore, the roadmap is composed of each method in the MCP list with its highest suspiciousness score. Methods that appear in more than one pair in the MCP list are included only once in the roadmap. Thus, using the ICD roadmap avoids repeatedly checking methods.

4.2.3 Filtering strategies to reduce code inspection

CH and ICD provide a roadmap to search for bugs in the most suspicious methods. For each method, a developer should also inspect its most suspicious blocks. We propose two strategies that aim to reduce the number of blocks inspected when a particular method of the roadmap is investigated. Therefore, both strategies are applied at block-level. These filtering strategies are called Level Score (LS) and Fixed Budget (FB).

The main idea of LS and FB is to avoid the inspection of blocks that are less likely to contain faults, thus reducing the effort needed to use the roadmap. Moreover, the filtering strategies were devised to mimic developers’ behavior while inspecting for bugs since they may not check all hints. Top hints are more likely to be verified. Both strategies vary according to the scores of the blocks in each method. Some methods may contain a large amount of LOCs being tedious and imprecise to decide how much code inside a method to inspect before moving to the following method. Even methods with reduced size may only contain a few blocks with higher suspiciousness scores, and so avoiding the other lower ranked blocks may improve the code inspection.

These filtering strategies act as criteria for the inspection of blocks inside the methods. However, we do not propose a single threshold value to apply to all methods as, for example, a single suspiciousness score value. Therefore, both filtering strategies can be applied with different ranking metrics no matter how they assign suspiciousness scores to the code elements. For instance, Ochiai often assigns suspiciousness scores to code elements with large distance among them while Tarantula assigns values with small differences.

To propose these filtering strategies, we also regarded that developers are frequently impatient and may abandon a technique if the bug is not found soon. Thus, strategies should effectively narrow down the amount of code to inspect. In doing so, they reduce the effort needed to find bugs and keep the developers’ interest.

Level score

Level Score (LS) is a strategy that deems each unique suspiciousness score of blocks inside a method as a level. A method usually contains blocks with different suspiciousness scores. One or more blocks may have the same score.

Level score was based on the concept breadth-first search. Vessey [1985] suggests that experienced developers use a breadth-first approach while debugging. Thus, a fault inspection can be improved using a more breadth inspection of methods and blocks through the roadmap instead of an in-depth search through blocks of each roadmap’s method.

Consider LS(l) in which l is the $l^{th}$ distinct suspiciousness score level of the blocks inside a method in descending order. All blocks with a score equal to or higher than the $l$ score should be investigated. As $l$ increases, more levels with lower scores are included and, thus, more blocks must be inspected.

For $l = 1$ (LS(1)), only blocks with the highest score are inspected. If the bug is not found among them, the following method should be inspected. For LS(2), all blocks with the highest suspiciousness score, and all blocks with the second highest suspiciousness score will be examined. Blocks with lower scores will be ignored, and, if the fault is not found, the following method should be investigated. The same reasoning is valid for other $l$ values.
Fixed budget

The Fixed Budget (FB) strategy aims at determining a maximum number of blocks (b) to inspect, regardless of how many methods will be inspected. It is based on the idea of representing an effort value that a developer would be willing to spend to find a bug.

Fixed budget assumes that an absolute number of code elements should be inspected to search for a bug without decreasing the developers’ interest [Parnin and Orso, 2011]. This effort budget may vary a lot between developers, and also may vary according to the severity of the fault—a severe bug tends to increase such a willingness.

To perform this strategy, we first use a general block list obtained from block spectra (i.e., before grouping blocks by their methods) in descending order of suspiciousness. Consider FB(b), where b is the maximum number of blocks to be inspected. To determine the blocks of FB(b), we verify the suspiciousness score of the bth block in the block list—the budget score.

If the number of blocks with a score equal to or greater than the budget score exceeds b, the score immediately above is used. The idea behind it is to avoid an over-budget. The only exception is when the budget score of the bth block is the highest score on the list. In this case, if the number of blocks exceeds b, we prefer to select an over-budget number of blocks to be inspected instead of an empty list.

After assigning the budget score, blocks are grouped by their respective methods. Following the roadmap, all blocks with scores equal to or greater than the budget score should be inspected. FB is directly affected by ties in the block list—if the score of the bth block is present in other blocks, the score above will be chosen, leading to less than b blocks to be inspected, except for the highest score. Small bs lead to few methods to be inspected since only methods that contain blocks within the budget score are inspected.

Suppose a b = 5 (FB(5)), the first two most suspicious blocks have scores equal to 1.0, the following three blocks have scores equal to 0.99, and the sixth block has a score of 0.98. The suspiciousness score of the fifth block will be the budget score, which is 0.99. This score is used to inspect the blocks of the methods indicated by the roadmap.

4.2.4 Debugging with roadmaps

We illustrate these strategies below with an example from a program used in our experiments. Consider a developer using a roadmap to search for the PJH_AK_1 bug of the Ant program (described in Section 4.3.1). Figure 4.2 shows the code excerpt containing the bug. It was seeded for experimental purposes [Do et al., 2005]. In the figure, the C preprocessor command #ifdef PJH_AK_1 was used to control the inclusion of the buggy line 112, instead of the correct line 114.

Table 4.1 shows the CH roadmap using Ochiai. The parse() method, which contains the faulty block, is the second method of the roadmap; its suspiciousness score is 1.0. Table 4.2 shows the ICD roadmap using Ochiai for the same fault. For ICD, the faulty method was classified in the first position. In both tables, the faulty method is highlighted in gray. The roadmaps were created following the guidelines described in Sections 4.2.1 and 4.2.2.

Table 4.3 shows parse()'s block list for the PJH_AK_1 bug. The Pos column has the block position in the general block list. The Id column has the block id. The following columns mean, respectively, the first line of the block, the source code of the first line, and the suspiciousness score. The first three blocks of parse() (the first three lines of Table 4.3) have the highest suspiciousness scores, but the fault site is not located in them. Thus, a strategy which allows us to investigate a few more blocks without excessively increasing the number of blocks examined in each method improves the ability of CH and ICD to search for bugs at block-level.

For LS equal to one (LS(1)), and using either a CH or an ICD roadmap, only the most suspicious blocks of each method will be inspected. In this case, the bug will be missed because the three blocks of parse() with highest suspiciousness scores do not include the bug. If LS is two (LS(2)), only one extra block will be inspected in parse(), which is the faulty block.
Since `resolveEntity()` is the most suspicious method of the CH roadmap, nine more blocks from this method will be investigated before the four blocks of `parse()` using LS(2), resulting in a total of thirteen blocks investigated until reaching the faulty one. Using the ICD roadmap, only the four blocks of `parse()` will be examined. In Table 4.3, the lines highlighted in gray are the blocks belonging to the inspection range for LS(2).

Suppose a developer opted to use FB and selected a budget of 10 blocks (FB(10)). The block list, including blocks of all methods for PJH_AK_1, is shown in Table 4.4. The gray lines are blocks from the `parse()` method. The score of the tenth block is 0.5. However, there are more than ten blocks with this score, which exceeds the budget. Thus, the budget score will be 0.57, i.e., the score immediately above. Only blocks with suspiciousness scores equal to or greater than 0.57 will be examined.

Using the CH roadmap, two blocks of the `resolveEntity()` method are inspected (in Table 4.4, these blocks are in positions four and five). After that, four blocks of `parse()` will be inspected; a total of six blocks are inspected until reaching the bug. Using the ICD roadmap, only four blocks of `parse()` will be inspected before the bug is reached.

Using both roadmaps and the filtering strategies for PJH_AK_1, in the worst case the developer will inspect thirteen blocks before reaching the fault site. In the best case, only four blocks will be inspected. As a result, the bug is in the set of suspicious blocks of `parse()` without excessively increasing the number of blocks to be examined.

### 4.2.5 CH and ICD complexity

To understand the execution costs of CH and ICD, we present an analysis of their time and space complexities. As for the time complexity, CH is built based on block spectrum. Each block is associated with its method during the instrumentation. Therefore, block and method spectra are gathered together, resulting in a complexity of $O(N)$, where $N$ is the number of blocks executed by the test suite. Regarding space complexity, CH stores the spectra of each block. Being $B$ the total of blocks, the space needed to store block spectra costs $O(B)$. It also requires an additional space to store the method information, which is proportional to the number of methods ($M$) of a program. This extra space for CH is quite small in comparison with the space needed to store block spectra, which entails a space complexity of $O(B + M) = O(B)$.

ICD is based on MCP and block spectra. To collect MCP spectra, we need to instrument the entry block and all possible exit blocks of each method. These blocks are also covered by the block-
Table 4.1: CH roadmap of the PJH_AK_1 bug

<table>
<thead>
<tr>
<th>Method</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>resolveEntity()</td>
<td>1.00</td>
</tr>
<tr>
<td>parse()</td>
<td>1.00</td>
</tr>
<tr>
<td>BuildException()</td>
<td>0.37</td>
</tr>
<tr>
<td>getMessage()</td>
<td>0.14</td>
</tr>
<tr>
<td>getPriority()</td>
<td>0.09</td>
</tr>
<tr>
<td>setName()</td>
<td>0.09</td>
</tr>
<tr>
<td>setDefaultTarget()</td>
<td>0.09</td>
</tr>
<tr>
<td>setUserProperty()</td>
<td>0.09</td>
</tr>
<tr>
<td>addBuildListener()</td>
<td>0.09</td>
</tr>
<tr>
<td>fireMessageLoggedEvent()</td>
<td>0.09</td>
</tr>
<tr>
<td>access$202()</td>
<td>0.09</td>
</tr>
<tr>
<td>access$400()</td>
<td>0.09</td>
</tr>
<tr>
<td>access$300()</td>
<td>0.09</td>
</tr>
<tr>
<td>access$100()</td>
<td>0.09</td>
</tr>
<tr>
<td>configureProject()</td>
<td>0.09</td>
</tr>
<tr>
<td>setDocumentLocator()</td>
<td>0.09</td>
</tr>
</tbody>
</table>

Table 4.2: ICD roadmap of the PJH_AK_1 bug

<table>
<thead>
<tr>
<th>Method</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>parse()</td>
<td>1.00</td>
</tr>
<tr>
<td>BuildException()</td>
<td>1.00</td>
</tr>
<tr>
<td>resolveEntity()</td>
<td>0.44</td>
</tr>
<tr>
<td>log()</td>
<td>0.44</td>
</tr>
<tr>
<td>access$300()</td>
<td>0.44</td>
</tr>
<tr>
<td>access$400()</td>
<td>0.44</td>
</tr>
<tr>
<td>set()</td>
<td>0.37</td>
</tr>
<tr>
<td>setMessage()</td>
<td>0.37</td>
</tr>
<tr>
<td>createTask()</td>
<td>0.35</td>
</tr>
<tr>
<td>Echo()</td>
<td>0.35</td>
</tr>
<tr>
<td>execute()</td>
<td>0.35</td>
</tr>
<tr>
<td>createAttributeSetter()</td>
<td>0.33</td>
</tr>
<tr>
<td>IntrospectionHelper$15()</td>
<td>0.33</td>
</tr>
<tr>
<td>fireMessageLoggedEvent()</td>
<td>0.14</td>
</tr>
<tr>
<td>getMessage()</td>
<td>0.14</td>
</tr>
<tr>
<td>executeTarget()</td>
<td>0.13</td>
</tr>
</tbody>
</table>

level instrumentation. Thereby, both MCP and block spectra are gathered together. Regarding the time complexity, ICD also depends on the number of executed blocks \(O(N)\). For the space complexity, ICD stores MCP spectra, which is linear with the number of caller-callee pairs \(P\) of a program run. ICD also stores block spectra \(B\). In the worst case scenario, all methods call and are called by all methods \(M^2\), i.e., \(P \leq M^2\), which entails a total space complexity of \(O(B + M^2)\).

The above analysis suggests that the extra space for CH is negligible and that ICD is a more space demanding technique. Table 4.5 shows the average memory space overhead to calculate the CH and ICD roadmaps in comparison with a single block list (BL) for all the programs evaluated in our experiments (see Section 4.3.1). Memory space overhead is the ratio between the memory size between CH, ICD, and BL. Such data was measured using a PC with an Intel Core i5 processor, 4GB of RAM running Ubuntu 13.04. CH had an average space overhead of 0.78% compared with BL, which confirms the estimate that the CH overhead is insignificant. ICD had an average overhead of 57% compared with BL. Though a sizable memory increase, it is far from the worst-case scenario.

Thus, for the real programs used in the evaluation, even considering that ICD has significant overhead, both the time and the space required to compute the roadmaps is affordable for current commodity computers.

4.2.6 Road2Fault tool

We developed an open source command line tool called Road2Fault\(^1\) to support the application of the proposed techniques. Road2Fault receives as input program spectra of two types: block and MethodCallPair (MCP). Road2Fault provides several ranking metrics to calculate fault suspiciousness, such as Ochiai and Tarantula. The Road2Fault’s outputs are the CH and ICD roadmaps and the list of most suspicious blocks.

To collect block and MCP spectra, Road2Fault uses the Instrumentation Strategies Simulator\(^2\) (InSS) [de Araujo et al., 2011] – a framework that allows rapid implementation of program instrumentation strategies in Java. Strategies to collect block and MCP spectra were implemented using InSS.

\(^1\)github.com/saeg/road2fault
\(^2\)github.com/saeg/inss
Figure 4.3 describes how InSS and Road2Fault work to produce the roadmaps and the block list. The program being debugged executes its JUnit\(^3\) test suite along with InSS, which tracks blocks and MCPs. When the execution finishes, block spectrum, MCP spectrum, and the result (pass or fail) for each test case are delivered to Road2Fault. We used InSS because it simplifies the collection of new spectrum such as MCP; however, any other tool providing the same information can be utilized in its place. In particular, there are many tools that provide block spectrum, such as CodeCover\(^4\) or JaCoCo\(^5\).

Road2Fault uses block spectrum, MCP spectrum, and the test results to generate a coverage matrix similar to the one shown in Table 2.4. It represents the program elements executed by each test case with their result. Then, Road2Fault calculates the suspiciousness scores for each program element according to the chosen metric and the roadmaps according to the algorithms presented in Sections 4.2.1 and 4.2.2.

---

Table 4.3: Block list of the PJH_AK_1 bug for parse()

<table>
<thead>
<tr>
<th>Pos</th>
<th>Id</th>
<th>Line</th>
<th>Code</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>337</td>
<td>149</td>
<td>catch(FileNotFoundException exc) {</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>359</td>
<td>156</td>
<td>if (inputStream != null) {</td>
<td>1.00</td>
</tr>
<tr>
<td>3</td>
<td>364</td>
<td>156</td>
<td>try{</td>
<td>1.00</td>
</tr>
<tr>
<td>4</td>
<td>68</td>
<td>112</td>
<td>uri = uri.substring(0, index) + &quot;%24&quot; ...; (=) bug</td>
<td>0.57</td>
</tr>
<tr>
<td>5</td>
<td>102</td>
<td></td>
<td>FileInputStream inputStream = null;</td>
<td>0.09</td>
</tr>
<tr>
<td>6</td>
<td>118</td>
<td></td>
<td>inputStream = new FileInputStream(buildFile);</td>
<td>0.09</td>
</tr>
<tr>
<td>7</td>
<td>158</td>
<td></td>
<td>inputStream.close();</td>
<td>0.09</td>
</tr>
<tr>
<td>8</td>
<td>162</td>
<td></td>
<td>}</td>
<td>0.09</td>
</tr>
<tr>
<td>9</td>
<td>106</td>
<td></td>
<td>SAXParser saxParser = getParserFactory();</td>
<td>0.09</td>
</tr>
<tr>
<td>10</td>
<td>110</td>
<td></td>
<td>for(int index = uri.indexOf(&quot;#&quot;); index != -1; ...{</td>
<td>0.09</td>
</tr>
<tr>
<td>11</td>
<td>123</td>
<td></td>
<td>}</td>
<td>0.0</td>
</tr>
<tr>
<td>12</td>
<td>239</td>
<td></td>
<td>String entitySystemId = path;</td>
<td>0.50</td>
</tr>
<tr>
<td>13</td>
<td>245</td>
<td></td>
<td>while (index != -1) {</td>
<td>0.50</td>
</tr>
<tr>
<td>14</td>
<td>246</td>
<td></td>
<td>}</td>
<td>0.50</td>
</tr>
<tr>
<td>15</td>
<td>265</td>
<td></td>
<td>return null;</td>
<td>0.50</td>
</tr>
<tr>
<td>16</td>
<td>259</td>
<td></td>
<td>}</td>
<td>0.50</td>
</tr>
</tbody>
</table>

Table 4.4: Block list of the PJH_AK_1 bug for all methods

<table>
<thead>
<tr>
<th>Pos</th>
<th>Id</th>
<th>Line</th>
<th>Code</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>337</td>
<td>149</td>
<td>catch(FileNotFoundException exc) {</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>359</td>
<td>156</td>
<td>if (inputStream != null) {</td>
<td>1.00</td>
</tr>
<tr>
<td>3</td>
<td>364</td>
<td>156</td>
<td>try{</td>
<td>1.00</td>
</tr>
<tr>
<td>4</td>
<td>248</td>
<td>259</td>
<td>} catch (FileNotFoundException fne) {</td>
<td>1.00</td>
</tr>
<tr>
<td>5</td>
<td>284</td>
<td>265</td>
<td>return null;</td>
<td>1.00</td>
</tr>
<tr>
<td>6</td>
<td>68</td>
<td>112</td>
<td>uri = uri.substring(0, index) + &quot;%24&quot; ...; (=) bug</td>
<td>0.57</td>
</tr>
<tr>
<td>7</td>
<td>226</td>
<td></td>
<td>project.log(&quot;resolving systemId: &quot; + systemId,...);</td>
<td>0.50</td>
</tr>
<tr>
<td>8</td>
<td>239</td>
<td></td>
<td>String entitySystemId = path;</td>
<td>0.50</td>
</tr>
<tr>
<td>9</td>
<td>245</td>
<td></td>
<td>while (index != -1) {</td>
<td>0.50</td>
</tr>
<tr>
<td>10</td>
<td>250</td>
<td></td>
<td>}</td>
<td>0.50</td>
</tr>
<tr>
<td>11</td>
<td>256</td>
<td></td>
<td>InputSource inputSource = new InputSource(...);</td>
<td>0.50</td>
</tr>
<tr>
<td>12</td>
<td>229</td>
<td></td>
<td>String path = systemId.substring(5);</td>
<td>0.50</td>
</tr>
<tr>
<td>13</td>
<td>234</td>
<td></td>
<td>while (index != -1) {</td>
<td>0.50</td>
</tr>
</tbody>
</table>

---

\(^3\)junit.org
\(^4\)codecover.org
\(^5\)eclemma.org/jacoco
Table 4.5: Average of memory size overhead of CH and ICD compared with BL

<table>
<thead>
<tr>
<th>Program</th>
<th>CH/BL (%)</th>
<th>ICD/BL (%)</th>
<th>ICD/CH (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ant</td>
<td>0.57</td>
<td>36.76</td>
<td>35.98</td>
</tr>
<tr>
<td>Commons Math</td>
<td>0.56</td>
<td>67.41</td>
<td>66.47</td>
</tr>
<tr>
<td>HSQLDB</td>
<td>0.15</td>
<td>68.50</td>
<td>68.24</td>
</tr>
<tr>
<td>JTopas</td>
<td>2.53</td>
<td>44.58</td>
<td>41.01</td>
</tr>
<tr>
<td>PMD</td>
<td>0.63</td>
<td>58.11</td>
<td>57.12</td>
</tr>
<tr>
<td>XML Security</td>
<td>0.90</td>
<td>35.20</td>
<td>33.99</td>
</tr>
<tr>
<td>XStream</td>
<td>0.16</td>
<td>88.81</td>
<td>88.51</td>
</tr>
<tr>
<td>Average</td>
<td><strong>0.78</strong></td>
<td><strong>57.05</strong></td>
<td><strong>55.90</strong></td>
</tr>
</tbody>
</table>

Figure 4.3: Steps to create the CH and ICD roadmaps

4.3 Experimental evaluation

We carried out an experiment to evaluate the CH and ICD roadmaps combined with the filtering strategies LS and FB. We compared the contextualization techniques with block list (BL). BL represents the existing SFL techniques, in which the suspiciousness list comprises more fine-grained level elements, such as blocks or statements. The blocks are ordered by their suspiciousness scores. The experiment evaluates the effectiveness of the techniques in locating faults, i.e., the amount of code to inspect to reach the faults.

Next, we present details of the experiment. We present the subject programs, treatments assessed, measurements, and data collection procedure.

4.3.1 Subject programs

We selected seven medium- to large-sized open source Java programs from various application domains: Ant, Commons Math, HSQLDB, JTopas, PMD, XML Security, and XStream. Ant is a tool for building applications, especially for Java projects. Commons Math is a library composed of mathematical and statistical functions. HSQLDB is a relational database. JTopas is a library for parsing arbitrary text data. PMD is a source code analyzer available for several programming languages. XML Security is a component library implementing XML signature and encryption standards. Finally, XStream is a library to (de)serialize objects (from)to XML.

Ant, JTopas, and XML Security were obtained from the Software-artifact Infrastructure Repository\(^6\) (SIR) [Do et al., 2005], which contains programs prepared for experimentation. The SIR program bugs were seeded. A faulty version of XStream\(^7\) was created with the same bug seeded by Gouveia et al. [2013]. Commons Math\(^8\), HSQLDB\(^9\), and PMD\(^10\) contain real bugs extracted from

\(^6\)sir.unl.edu/portal/index.php
\(^7\)x-stream.github.io/repository.html
\(^8\)commons.apache.org/proper/commons-math
\(^9\)hsqldb.org
\(^10\)pmd.github.io
their repositories. We identified twenty, four, and two real faults in Commons Math, HSQLDB, and PMD, respectively. All versions of the programs used in the experiment contain a single bug.

Table 4.6 presents the characteristics of the selected programs. The columns indicate, respectively: the name of the program; the size variation of the faulty versions in thousands of lines of code (KLOC); the number of versions; the number of faults; the fault type, in which R means real faults and S means seeded faults; and the variation of the test suite size present in the different versions of the programs.

<table>
<thead>
<tr>
<th>Program</th>
<th>KLOC</th>
<th>Versions</th>
<th>Faults</th>
<th>Real</th>
<th>Test Cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ant</td>
<td>25–80</td>
<td>8</td>
<td>18</td>
<td>S</td>
<td>141–851</td>
</tr>
<tr>
<td>Commons Math</td>
<td>16–39</td>
<td>3</td>
<td>20</td>
<td>R</td>
<td>1167–2114</td>
</tr>
<tr>
<td>HSQLDB</td>
<td>155–158</td>
<td>2</td>
<td>4</td>
<td>R</td>
<td>1146–1299</td>
</tr>
<tr>
<td>JTopas</td>
<td>1.8–5</td>
<td>3</td>
<td>4</td>
<td>S</td>
<td>22–31</td>
</tr>
<tr>
<td>PMD</td>
<td>50</td>
<td>1</td>
<td>2</td>
<td>R</td>
<td>616</td>
</tr>
<tr>
<td>XML Security</td>
<td>16–18</td>
<td>3</td>
<td>13</td>
<td>S</td>
<td>84–94</td>
</tr>
<tr>
<td>XStream</td>
<td>17</td>
<td>1</td>
<td>1</td>
<td>S</td>
<td>1457</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>—</td>
<td>—</td>
<td>62</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

### 4.3.2 Treatments

The treatments were devised to assess the effectiveness of the contextualization techniques in finding bugs. The effectiveness was measured by the number of faults found using the techniques, and by the number of blocks inspected until finding the faults. We used a block-level suspiciousness list (BL) as the baseline for the comparison. Two metrics were used to compare the techniques: Ochiai and Tarantula. Both CH and ICD use the filtering strategies LS and FB for inspection at block-level.

The experiment was also designed to verify the number of methods investigated to locate a bug. Our intuition is that locating bugs by investigating a limited number of methods may incentivize the use of automated SFL techniques.

**CH and ICD roadmaps.** Each technique generates its roadmap. However, the procedure to examine the roadmaps is the same: a developer using a roadmap will start by investigating the most suspicious method. Inside the method, s/he will investigate the most suspicious blocks using one of the filtering strategies, LS or FB.

**CH-FB(b).** This is the CH roadmap along with FB for a b value.

**CH-LS(2).** This is the CH roadmap along with LS for a level 2. The same rationale is applied to ICD-LS(2) and ICD-FB(b).

**Block list.** In this treatment, referred to simply as BL, the developer starts the search by inspecting blocks with the maximum suspiciousness scores. In failing to discover the faulty block, s/he proceeds to the next score, and so forth until reaching the faulty block. The BL treatment is given by the number of blocks inspected in all scores, including the faulty block score.

The number of blocks with the same suspiciousness score varies for the different faults investigated. To calculate the number of blocks inspected, we count all blocks with a suspiciousness score greater than or equal to the faulty block. Thus, we evaluate the worst case scenario for all treatments, in which all blocks with the same suspiciousness are verified.
4.3.3 Procedure

In this section, we present the procedures used to collect the data on the effectiveness of the CH-LS(2), CH-FB, ICD-LS(2), ICD-FB, and BL techniques.

Effort budget

We assessed the techniques by measuring their effectiveness to rank the faults within the effort budgets. We estimate the effort budget as a fixed amount of blocks to be inspected by a developer, independent of the program size. The intuition is that if s/he is unable to locate the bug in this fixed number of blocks, the developer will resort to another fault localization strategy [Parnin and Orso, 2011]. Steimann et al. [2013] also argued that the use of absolute measures reflects a more realistic effort for fault localization. In practice, a technique that narrows down a bug to 1% of the code—as for EXAM score [Wong et al., 2010]—for a particular program $P$ still demands the inspection of 1K LOC if $P$ has 100K LOC. That is too much code to inspect! The effort budget can also be considered as a measure of efficiency—the fault not only must be found, it also must be classified within a limited number of elements.

Suppose that a fault is located by inspecting 19 and 57 blocks using ICD-LS(2) and BL, respectively. In this case, the bug can be found within an effort budget of 20 inspected blocks using ICD-LS(2). However, using only BL, the same budget is not enough, since it requires the inspection of 57 blocks. Nonetheless, with an effort budget of 70 blocks, this fault can be found by both techniques. We collected data for fifteen different budgets, namely, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 60, 70, 80, 90, and 100 blocks.

Creation of roadmaps and lists

First, we identified the faults that caused observable failures during test execution. In total, sixty-two faults met this requirement and were used in the experiment, as shown in Table 4.6. MCP and block spectra were collected for each test case, as well as their outcomes. Then, the roadmaps and the lists of suspicious blocks were created by the Road2Fault tool using Ochiai and Tarantula ranking metrics.

Since each method in a JUnit test class is a test case, failing and passing test cases were determined by verifying the output of each method. We deemed test cases that failed or raised an exception during the execution as the failing ones; otherwise, we considered them as passing test cases. Some test cases fail even when executed in the original version (without bugs) of the programs. We excluded these test cases since their failures are unrelated to the bugs being investigated.

For HSQLDB, we excluded one of the test classes, which contains 181 test cases. Due to InSS limitations, this class took too much time to execute when instrumented. However, no failures were revealed by these test cases when we executed the test suite in the faulty versions without InSS. In PMD, two test cases were excluded because they represent faults not in the program code, but rather in the test data (XML files with faulty Java code). For XStream, we excluded 10 test classes that conflicted with libraries, a total of 57 test cases, which could not instantiate abstract classes, used mock libraries or failed to deal with threads.

Multiple-fault versions

To evaluate the performance of our techniques in programs containing multiple faults, we combined faults from the single-fault versions of the subject programs. We generated 62 multiple-fault versions to keep the same number of single-fault versions. Table 4.7 shows the number of multiple-fault versions. The All column shows the sum of 2-fault and 4-fault versions of each program.

We created 4-fault versions whenever possible, for program versions with 4 or more faults. Otherwise, we created 2-fault versions. For versions with a large number of faults (e.g., Commons Math has a version containing 12 faults), we randomly generated multiple-fault versions ensuring
that each fault was used at least once. Moreover, all faults that we used are placed in single lines to avoid different chances of finding each bug [Abreu et al., 2009b; Lucia et al., 2014].

Table 4.7: Multiple-fault versions

<table>
<thead>
<tr>
<th>Program</th>
<th>2-fault</th>
<th>4-fault</th>
<th>All</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ant</td>
<td>16</td>
<td>2</td>
<td>18</td>
</tr>
<tr>
<td>Commons Math</td>
<td>3</td>
<td>21</td>
<td>24</td>
</tr>
<tr>
<td>HSQLDB</td>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>JTopas</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>PMD</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>XML Security</td>
<td>1</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>25</strong></td>
<td><strong>37</strong></td>
<td><strong>62</strong></td>
</tr>
</tbody>
</table>

To evaluate the fault localization effectiveness in the multiple-fault versions, we deem the number of blocks inspected until reaching the first fault [Pearson et al., 2017; Wong et al., 2012; Yu et al., 2008]. The rationale is that a developer tends to inspect a list from the most suspicious elements. If s/he locates the well-ranked fault, s/he would fix it and rerun the tests, which would change the suspiciousness scores of remaining faults. Thus, s/he would repeat the process until there are no more failures. This scenario simulates a developer that tries to locate one bug at a time.

**Effectiveness data collection**

The Road2Fault tool has functions to collect the effectiveness data, analyzing the roadmaps and block lists according to the treatments presented above. This approach avoids the need for manual intervention, as it is less prone to evaluation errors. We collected the number of blocks and methods inspected until reaching each fault for the five treatments.

4.4 Results and discussion

In this section, we present the results of the experiments by evaluating the effectiveness of CH-LS(2), CH-FB, ICD-LS(2), ICD-FB, and BL. The results and discussion are guided by the research questions introduced in Section 4.1.

In the following figures, we present the data obtained in the experiments. The bars were organized following the legends’ order. CH-LS(2) and CH-FB are represented, respectively, by black and green bars. ICD-LS(2) and ICD-FB, are represented, respectively, by red and blue bars. The baseline SFL technique, Block List (BL), is represented by light gray bars. Hereafter, whenever we mention CH and ICD, we refer to the use of CH and ICD roadmaps, respectively.

4.4.1 RQ1: Are roadmaps combined with filtering strategies effective for fault localization?

Figures 4.4 and 4.5 show the aggregated effectiveness values considering all programs for Ochiai and Tarantula, respectively. The comparison between the roadmaps and BL was carried out to verify not only which technique reaches more faults, but also which technique finds them by inspecting less blocks.

For Ochiai (Figure 4.4), ICD using the FB filtering strategy found 21 faults for the tightest budget (ICD-FB(5)). This means that 33.87% of the faults are found by inspecting at most five blocks. BL found 14 faults, one more than CH using LS(2) (CH-LS(2)). ICD-FB hit more faults than the other techniques for all budgets, especially in lower ranges. CH-FB also reached more faults than CH-LS(2), ICD-LS(2) and BL. CH-LS(2) hit more faults than BL within the budget ranges between 10 and 50. ICD-LS(2) had the worst performance among the techniques.
Using Tarantula (Figure 4.5), ICD-FB hit more faults in the ranges 5–35, except for budget 20. CH-FB and CH-LS(2) were more closely to ICD-FB, with some draws. CH-FB overcame the other techniques from budget 60 to 100. All treatments obtained better results using Ochiai than using Tarantula for all budgets. This fact is not a surprise since several studies have shown that Ochiai outperforms Tarantula [Abreu et al., 2007; Le et al., 2013a; Ma et al., 2014; Xie et al., 2013]. It is worthy to note, though, that CH and ICD improve both Ochiai and Tarantula results; that is, they are orthogonal to the current SFL techniques.

ICD-FB hit 46 faults by inspecting up to 30 blocks using Ochiai, which represents 74.19% of all faults. To find a similar number of faults with BL, we need to inspect up to 50 blocks, i.e., 66.67% more blocks than ICD-FB. Taking into account only the 52 faults located by any technique at the highest budget, ICD-FB found 88.46% of the located faults by inspecting up to 30 blocks. CH-FB reached 72.58% of all faults and 86.54% of all located faults, while BL reached 66.13% and 78.85%, respectively, for the same budget. Thus, the contextualization techniques perform particularly well for small budgets in comparison to block list. This difference may be the tipping point between success or failure of an SFL technique, especially if the developer allocates tighter budgets.

![Graph showing effectiveness of treatments using various budgets for Ochiai](image)

**Figure 4.4:** Effectiveness of treatments using various budgets for Ochiai

The programs used in our evaluation have different characteristics in regarding the number of LOCs, test suite size, program domain, coupling, and so on. Some programs have a limited number of faults. However, the behavior of the treatments was similar among all programs.

Tables 4.8 and 4.9 show the medians of blocks inspected by technique until reaching the bugs, respectively for Ochiai and Tarantula, for all programs and also for each program separately. The bold values are the cases in which the faults were better ranked by the techniques. ICD-FB obtained the lowest medians using Ochiai and Tarantula considering all programs. Regarding each program, either ICD-FB or CH-FB achieved the lowest medians, except for HSQLDB.

We conducted a statistical analysis to compare the performance of our techniques against BL. We applied the Anderson-Darling normality test [Anderson and Darling, 1954] to check the distribution our data. This test confirmed that the distribution of our data is not normal.

We compared the number of blocks inspected by the techniques to reach the faults for each budget, which we call Statistical Test 1 ($ST_1$). We used the Wilcoxon signed-rank test [Wilcoxon, 1945] with a confidence level of 95%. The Wilcoxon signed-rank test assesses differences in the

---

11We use the median for the comparison due to the non-normal distribution of our data.
## Results and Discussion

### Table 4.8: Median of the ranking position of the bugs using Ochiai

<table>
<thead>
<tr>
<th>Program</th>
<th>CH-LS(2)</th>
<th>ICD-LS(2)</th>
<th>CH-FB</th>
<th>ICD-FB</th>
<th>BL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ant</td>
<td>10.5</td>
<td>16.5</td>
<td>9</td>
<td>6.5</td>
<td>19.5</td>
</tr>
<tr>
<td>Commons Math</td>
<td>8</td>
<td>10</td>
<td>8.5</td>
<td>7</td>
<td>12</td>
</tr>
<tr>
<td>HSQLDB</td>
<td>65</td>
<td>101</td>
<td>101</td>
<td>101</td>
<td>101</td>
</tr>
<tr>
<td>JTopas</td>
<td>24</td>
<td>60</td>
<td>18.5</td>
<td>22.5</td>
<td>32.5</td>
</tr>
<tr>
<td>PMD</td>
<td>55</td>
<td>57</td>
<td>53.5</td>
<td>53.5</td>
<td>53.5</td>
</tr>
<tr>
<td>XML Security</td>
<td>12</td>
<td>85</td>
<td>9</td>
<td>9</td>
<td>12</td>
</tr>
<tr>
<td>XStream</td>
<td>27</td>
<td>34</td>
<td>27</td>
<td>28</td>
<td>40</td>
</tr>
<tr>
<td><strong>All programs</strong></td>
<td><strong>11.5</strong></td>
<td><strong>32.5</strong></td>
<td><strong>9.5</strong></td>
<td><strong>8.5</strong></td>
<td><strong>17.5</strong></td>
</tr>
</tbody>
</table>

Table 4.8 shows the results of $ST_1$. The cells contain those budgets for which the null hypothesis was rejected. Whenever the cell is empty, the null hypothesis could not be rejected.

ICD-FB and CH-FB had the null hypothesis rejected for budgets from 10 to 100 using either Ochiai or Tarantula. Only ICD-FB also had statistical significance at budget 5 for Ochiai. We believe that the absence of statistical significance for the budget 5 occurs because this budget is very restrictive, with a large number of draws, reducing the sample size for statistical tests.

Using Ochiai, CH-LS(2) could not reject the null hypothesis for all budgets; though, the null hypothesis was rejected for some budgets (50, 80, 90, 100) using Tarantula. Thus, one cannot assume that CH-LS(2) inspects less blocks than BL significantly. ICD-LS(2) could not reject the null hypothesis for all budgets and the two ranking metrics. Therefore, a developer will tend to find bugs inspecting less code when s/he uses ICD-FB for all budgets and CH-FB for almost all budgets used to limit the inspection.

We also compared the number of faults found by the techniques for all budgets ($ST_2$). We used the Kolmogorov-Smirnov test [D’Agostino and Stephens, 1986] for this comparison, which is suitable for cumulative distribution data. We considered a confidence level of 95%. The null and alternative hypotheses are shown in Table 4.12.

Table 4.13 shows the statistical results (p-values) for this comparison. We reject the null hypothesis for CH-FB and ICD-FB using Ochiai, which means that these techniques found more faults in all programs.
Table 4.9: Median of the ranking position of the bugs using Tarantula

<table>
<thead>
<tr>
<th>Program</th>
<th>CH-LS(2)</th>
<th>ICD-LS(2)</th>
<th>CH-FB</th>
<th>ICD-FB</th>
<th>BL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ant</td>
<td>13</td>
<td>33.5</td>
<td>12.5</td>
<td>16</td>
<td>26.5</td>
</tr>
<tr>
<td>Commons Math</td>
<td>8</td>
<td>10</td>
<td>8.5</td>
<td>7</td>
<td>12</td>
</tr>
<tr>
<td>HSQLDB</td>
<td>101</td>
<td>101</td>
<td>101</td>
<td>101</td>
<td>101</td>
</tr>
<tr>
<td>JTopas</td>
<td>27</td>
<td>61</td>
<td>20</td>
<td>24.5</td>
<td>35.5</td>
</tr>
<tr>
<td>PMD</td>
<td>55</td>
<td>57</td>
<td>53.5</td>
<td>53.5</td>
<td>53.5</td>
</tr>
<tr>
<td>XML Security</td>
<td>78</td>
<td>101</td>
<td>54</td>
<td>101</td>
<td>101</td>
</tr>
<tr>
<td>XStream</td>
<td>27</td>
<td>34</td>
<td>27</td>
<td>28</td>
<td>40</td>
</tr>
<tr>
<td>All programs</td>
<td>18.5</td>
<td>52.5</td>
<td>18.5</td>
<td>17</td>
<td>28.5</td>
</tr>
</tbody>
</table>

Table 4.10: Test hypotheses for ST\(_1\)

\(H_0\) The compared techniques have the same effectiveness

\(H_1\) Our proposed techniques are more effective than BL

Table 4.11: Budgets with statistical significance for ST\(_1\)

<table>
<thead>
<tr>
<th>Metric</th>
<th>CH-LS(2) x BL</th>
<th>ICD-LS(2) x BL</th>
<th>CH-FB x BL</th>
<th>ICD-FB x BL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ochiai</td>
<td>—</td>
<td>—</td>
<td>10–100</td>
<td>5–100</td>
</tr>
<tr>
<td>Tarantula</td>
<td>50, 80–100</td>
<td>—</td>
<td>10–100</td>
<td>10–100</td>
</tr>
</tbody>
</table>

Table 4.12: Test hypotheses for ST\(_2\)

\(H_0\) The compared techniques reach the same number of faults through the budgets

\(H_1\) Our proposed techniques reach more faults than BL through the budgets
Table 4.13: P-values (%) for ST$_2$

<table>
<thead>
<tr>
<th>Metric</th>
<th>CH-LS(2) x BL</th>
<th>ICD-LS(2) x BL</th>
<th>CH-FB x BL</th>
<th>ICD-FB x BL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ochiai</td>
<td>33.42</td>
<td>100.00</td>
<td>3.18</td>
<td>3.18</td>
</tr>
<tr>
<td>Tarantula</td>
<td>33.42</td>
<td>100.00</td>
<td>9.07</td>
<td>9.07</td>
</tr>
</tbody>
</table>

than BL with statistical significance. There is no significant difference in the number of faults found through the budgets for the other comparisons. Also, BL found more faults than ICD-LS(2) with statistical significance. Therefore, the results suggest that ICD-FB and CH-FB improve the fault localization when compared to BL using a top performing ranking metric.

Regarding the research question, CH-FB and ICD-FB found more faults inspecting less blocks than BL with statistical significance. CH-LS(2) also reached more faults than BL, as shown in Figures 4.4 and 4.5, but without statistical significance. As a result, CH-FB and ICD-FB appear to be promising. Furthermore, CH is less computationally costly than ICD with a similar cost to BL.

Restricted budgets tend to be more realistic than larger ones [Parnin and Orso, 2011; Steimann et al., 2013]. ICD-FB, CH-FB, and CH-LS(2) located more bugs within restricted budgets, independent of the ranking metric (Ochiai or Tarantula) used. In practice, we do not know how much code a developer is willing to investigate when searching for bugs. Because each developer has his or her effort budget, it is safer to assume smaller budgets than larger ones. A technique that works best for such cases will more likely be used. ICD-FB, CH-FB, and CH-LS(2) seem to be a better choice than BL for these situations.

Thus, the experimental data suggest that CH, ICD and the filtering strategies LS and, especially, FB improve the effectiveness of SFL techniques. Also, they can guide developers performing debugging tasks. However, user studies are needed to verify these results in practice.

4.4.2 RQ2: Which is the best level (l) value to use in the Level Score filtering strategy?

We evaluated LS using different levels to understand which l values can reach more bugs. We used the same programs assessed in Section 4.3.1. Figures 4.6 and 4.7 show the number of faults found for different effort budgets, respectively for ICD and CH using five LS levels, from 1 to 5, and Ochiai.

For ICD and CH, most of the budgets reached more faults using LS(2). The exceptions are the tightest budget (5), in which LS(1) always performs better; LS(1) is also more effective inspecting up to 10 blocks for ICD; and some instances of higher budgets (from 80 to 100 for ICD and from 50 to 100 for CH), in which LS(3), LS(4), or LS(5) reaches more faults than LS(2). Levels from 3 to 5 find more faults in such cases because higher budgets allow the inspection of more blocks, which conversely leads to less faults found in more restricted budgets. Figures 4.8 and 4.9 show the results of this analysis for Tarantula, which presented a behavior similar to that of Ochiai.

The lower the level, the lesser the number of inspected blocks to reach a fault. Thus, it seems reasonable choosing a level that reaches more faults for most of the effort ranges and does not increase excessively the amount of blocks to inspect in the methods.

LS(1) is very restrictive and may result in finding too few faults. For cases in which a faulty block have the highest suspiciousness, LS(1) will reach the bug inspecting the lowest amount blocks possible. In many other cases, LS(1) will not reach the bug. Conversely, LS with l > 2 may lead to the inspection of too many blocks, increasing the chances of a developer giving up on using the technique. When the faulty block has the highest suspiciousness, LS(2) will reach it, but useless blocks may also be inspected. Faulty blocks with lower scores have more chance of being found as l increases. However, a good fault localization technique is supposed to classify faults among the higher scores.
We used LS(2) in the evaluation. This value seems reasonable in practice since only two levels of suspiciousness scores are investigated using such a strategy. Moreover, using LS(2) seems closer to the developer’s behavior. A developer investigating a method will check the most suspicious blocks, perhaps the second and third most suspicious ones, but not all blocks. The Level Score must be lower enough to avoid the inspection of an excessive number of blocks, but it should not be too restrictive to not include the fault.

4.4.3 RQ3: Which is the most helpful filtering strategy for fault localization?

Using FB, both ICD and CH had better results than BL with statistical significance. CH also exhibited better results using LS compared to BL. The combination between ICD and LS presented the worst results in the experiments.

FB is an aggressive strategy for small budgets (e.g., 15 blocks). Even for a budget of 10 blocks, CH-FB and ICD-FB reached more than half of the faults using Ochiai. Moreover, they found more faults than BL, which means that they help to reach additional faults, especially for small budgets.

We used LS with a value equal to 2, which means that inspecting two levels of suspiciousness scores is enough to find a large amount of the faults for the CH roadmap. Regarding the number of faults found, the results of CH-LS(2) had no statistical significance compared with BL. CH-LS(2) reached more faults than BL using Ochiai or Tarantula. Conversely, LS(2) and ICD were less effective than BL, which indicates that the ICD roadmap using LS(2) does not improve fault localization.

As a result, we conclude that **Fixed Budget has a better overall performance**. Its effectiveness is independent of the roadmap chosen by the developer. If s/he decides to use CH or ICD, the strategy of choice should be FB.

4.4.4 RQ4: How many methods are inspected to reach the faults using roadmaps and filtering strategies?

Figures 4.4 and 4.5 show the number of blocks inspected using the CH and ICD roadmaps until hitting the bug site. These blocks are spread across several methods. A valid question a developer using CH and ICD may pose is: how many methods belonging to a roadmap should I inspect to locate a bug?
4.4 RESULTS AND DISCUSSION

Figures 4.10 and 4.11 show the number of methods inspected to locate the bugs for Ochiai and Tarantula, respectively. For CH and ICD using LS, a fixed number of methods are inspected because the LS value is always 2. For CH and ICD using FB, each budget has its own number of inspected methods. For FB, we present the highest number of faults found through the different budgets.

In Figure 4.10, ICD-FB found 30 faults inspecting only the first method of the roadmap; and 28 and 27 faults were found by CH-FB and CH-LS(2), respectively. This means that ICD-FB classified the faulty method in the first position for 48.39% of all faults (62), and 57.69% of the faults that were found (52) using Ochiai. Furthermore, 70.97% of all faults were reached inspecting at most four methods using CH-FB, which represents 84.62% of the located faults. The techniques reached a great number of faults by inspecting only a few methods: more than 50% of all faults for ICD-FB, CH-FB, and CH-LS(2) after inspecting two methods. ICD-LS(2) found less faults, 35.48% of all faults after inspecting four methods.

The results for Tarantula (Figure 4.11) show that ICD-FB, CH-FB, and CH-LS(2) also reached a similar amount of bugs by visiting less methods: 19 faults were reached by inspecting one method. CH-FB found 36 faults by inspecting up to four methods. For all treatments, Ochiai reached more faults than Tarantula.

If a developer using CH with Ochiai is recommended to inspect at most four methods, s/he will locate around 70% of the bugs, independently of the filtering strategy. For this particular experiment, a small number of the roadmaps’ methods lead to the localization of a sizable amount of bugs. Thus, the number of methods can be used as a guide for debugging based on SFL techniques, since a developer tends to use a technique that will likely find bugs by inspecting only a few methods.

4.4.5 RQ5: Are roadmaps combined with filtering strategies effective for programs containing multiple faults?

Figures 4.12, 4.13, and 4.14 show the effectiveness results using Ochiai for all multiple-fault versions, 2-fault versions, and 4-fault versions, respectively. Considering all multiple-fault versions (Figure 4.12), ICD-FB and CH-FB reached more faults than BL, especially for small budgets. CH-LS(2) was slightly better than BL, and ICD-LS(2) had the worst performance. The results are similar to the ones obtained by looking at the 2-fault and 4-fault versions separately.
Figure 4.8: Faults found by ICD for different effort budgets and LS values using Tarantula

Table 4.14: Test hypotheses for \( ST_1 \)

\[ H_0 \] The compared techniques have the same effectiveness in multiple-fault programs
\[ H_1 \] Our proposed techniques are more effective than BL in multiple-fault programs

Figure 4.15 shows the results for all multiple-fault versions using Tarantula. All techniques performed worse compared to Ochiai. CH-LS(2) had better overall performance, and even ICD-LS(2) outperformed BL using Tarantula. Figures 4.16 and 4.17 show the results for the 2-fault and 4-fault versions using Tarantula.

As in Section 4.4.1, we performed a statistical analysis of the multiple-fault versions to compare (1) the number of blocks inspected by each technique to reach the first faults for each budget (\( ST_1 \)) and (2) the number of faults found by each technique for all budgets (\( ST_2 \)). The null (\( H_0 \)) and alternative (\( H_1 \)) hypotheses for \( ST_1 \) and \( ST_2 \) are shown in Tables 4.14 and 4.15, respectively.

Tables 4.16 and 4.17 show the statistical results for \( ST_1 \) and \( ST_2 \) regarding the multiple-fault versions, respectively. The All, 2, and 4 columns represent all multiple faults, 2-fault, and 4-fault versions, respectively. We can see that CH-FB and ICD-FB locate the first fault better than BL with statistical significance for almost all budgets. CH-LS(2) is better than BL for large budgets in the 2-fault versions using Ochiai and for all multiple-fault versions and budgets using Tarantula. For \( ST_2 \), CH-FB and ICD-FB can locate more faults than BL with statistical significance in most cases, while CH-LS(2) is better for the 2-fault versions using Ochiai and for all multiple-fault versions using Tarantula.

Overall, the results for multiple faults show that at least one of the faults is well-ranked by the techniques in most cases for all techniques, including BL, which resulted in a large number of faults found in the small budgets. **ICD-FB, CH-FB, and CH-LS(2) performed better than BL**

Table 4.15: Test hypotheses for \( ST_2 \)

\[ H_0 \] The compared techniques reach the same number of faults through the budgets in multiple-fault programs
\[ H_1 \] Our proposed techniques reach more faults than BL through the budgets in multiple-fault programs
4.5 Threats to validity

The discussion regarding threats to validity focuses on internal, external, and construct validities. The internal threats are the tools used to produce the block lists and roadmaps. The implementation of these tools was manually checked by applying them to small programs; the data collected in the experiment, however, were manually checked by sampling due to the size of the programs.

Regarding the external validity, we used programs from different domains (e.g., mathematics, database systems, text processing), and with fairly large sizes, to expose the techniques to a variety of contexts. However, the results presented in this paper cannot be generalized to other programs.

The construct validity relates to the suitability of our effectiveness metric. Effectiveness compares the ability of a technique to fit the bug site within a particular effort budget (from 5 to 100 blocks). Our choice of the effort budgets was arbitrary. However, our focus on lower budgets aims at replicating realistic scenarios for debugging techniques. Nevertheless, user studies should be performed to assess this issue.

Our experiment was built to evaluate how quickly a technique will reach the fault site. We caution, though, that reaching the bug site does not necessarily mean identifying a fault. It has

---

**Figure 4.9:** Faults found by CH for different effort budgets and LS values using Tarantula

**Table 4.16:** Budgets with statistical significance for $ST_1$ for multiple-fault versions

<table>
<thead>
<tr>
<th>Treatments</th>
<th>Ochiai</th>
<th></th>
<th>Tarantula</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>All</td>
<td>2</td>
<td>4</td>
<td>All</td>
</tr>
<tr>
<td>CH-LS(2) x BL</td>
<td>—</td>
<td>—</td>
<td>5–100</td>
<td>5–100</td>
</tr>
<tr>
<td>ICD-LS(2) x BL</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>CH-FB x BL</td>
<td>5–100</td>
<td>5–100</td>
<td>10–100</td>
<td>5–100</td>
</tr>
<tr>
<td>ICD-FB x BL</td>
<td>5–100</td>
<td>5–100</td>
<td>5–100</td>
<td>10–100</td>
</tr>
</tbody>
</table>

for almost all budgets. The prevalence of at least one fault observed in our results corroborates findings of previous work [DiGiuseppe and Jones, 2015]. Thus, our techniques performed well in programs with multiple faults. We caution that more effort is needed to locate the other existing faults.
been shown that the perfect bug detection [Parnin and Orso, 2011] assumption, which presumes that a developer will identify a faulty statement simply by inspecting it, is not always guaranteed in practice.

4.6 Final remarks

In this study, we presented techniques to provide contextual information for fault localization. Code Hierarchy (CH) and Integration Coverage-based Debugging (ICD) generate a list of methods (i.e., a roadmap) that are more likely to be faulty. Each method contains a list of its basic blocks to allow for code inspection at a fine-grained level inside the methods. We proposed two filtering strategies—Level Score (LS) and Fixed Budget (FB)—to limit the inspection inside the methods, which are used along with roadmaps to locate faults. Using LS or FB, the developer only needs to inspect the most suspicious blocks. Thus, contextualization establishes an order of inspection for the methods and which blocks to inspect in these methods.

We used the concept of effort budget to measure the effectiveness of the techniques during the fault localization process. An effort budget represents the number of blocks a developer investigates before abandoning a fault localization technique. This metric leads toward more practical evaluations of fault localization techniques by deeming amounts of code that are feasible for practical use.

The experiments carried out in this work used sixty-two faults from seven real open source programs containing from 2K to 158K lines of code. We compared the effectiveness of our techniques with a block list (BL).

ICD and CH using FB (ICD-FB and CH-FB) were more effective to find faults inspecting less
code than BL with statistical significance. Moreover, 70% of the faults were found by inspecting at most four methods using ICD-FB or CH-FB, and 67% using CH-LS. ICD-FB, CH-FB, and ICD-LS(2) were also more effective for programs containing multiple faults, especially with small budgets.

CH in association with FB seems to be a particularly good combination because it significantly improves the performance of Ochiai for small budgets at a negligible extra cost. This improvement may be the tipping point between success or failure of an SFL technique. We consider that techniques that reduce the absolute number of blocks/methods are more likely to be adopted in real development settings.

The results shown that our techniques improved the performance of SFL techniques (Ochiai and Tarantula), reducing the number of blocks inspected to find bugs. Moreover, the results suggest that the contextualization provided by roadmaps and filtering strategies is useful for guiding developers toward faults, bringing together information of suspicious methods and their related suspicious blocks. However, users studies may help to assess this issue.

In the next chapter, we present a user study that we carried out to understand how developers use SFL in practice.
Figure 4.12: Effectiveness of treatments for all multiple-fault versions using various budgets for Ochiai

Figure 4.13: Effectiveness of treatments for 2-fault versions using various budgets for Ochiai
Figure 4.14: Effectiveness of treatments for all 4-fault versions using various budgets for Ochiai

Figure 4.15: Effectiveness of treatments for all multiple-fault versions using various budgets for Tarantula
Figure 4.16: Effectiveness of treatments for 2-fault versions using various budgets for Tarantula

Figure 4.17: Effectiveness of treatments for 4-fault versions using various budgets for Tarantula
Chapter 5

User study

This study evaluates how developers use SFL through a practical assessment. We aim to understand how developers interact with an SFL tool and whether SFL improves their performance to locate bugs. We also evaluate the developers’ intention to use SFL in the future.

This chapter presents the details of our study design. We also describe the Jaguar SFL tool, which was used in the experiments. Then, we present the results, discussion, threats to validity and final remarks.

5.1 Research questions

In this study, we want to answer the following general question: Is SFL useful in practice? Our specific questions that address our concerns about the practical use of SFL are retaken below:

RQ6: Can SFL help more developers to locate faults?
RQ7: Can SFL help developers to locate faults faster?
RQ8: Can SFL guide more developers to inspect faulty code?
RQ9: Are the study participants intended to use SFL in practice?
RQ10: The level of code information impacts on the fault localization performance?

5.2 User study design

This section presents our user study setup. We replicated previous studies [Parnin and Orso, 2011; Xia et al., 2016; Xie et al., 2016] to investigate whether SFL can or cannot be useful in practice. As in previous work (see Section 3.2.5), we assessed if SFL improves the developers’ effectiveness and efficiency in debugging. We also present our findings regarding how developers interact with an SFL tool and their opinions about SFL.

Differently from previous work, we evaluate how closer to faulty code elements an SFL tool can take its users. More specifically, we assess if there is a correlation between reaching a fault site and finding a bug. Then, we assess if developers can identify the faulty methods and reach the faulty lines using SFL. In this way, an SFL tool can be useful to locate bugs or, at least, to provide hints for developers start to understand faulty behaviors. We also assess how the developers perceive the usefulness and ease of use of SFL in practice, which indicates their willingness to use SFL in the future. This study was approved by the Ethics committee of the Institute of Mathematics and Statistics (IME-USP). In what follows, we describe the methodology, techniques and procedures applied in this study.
5.2.1 Jaguar SFL tool

Jaguar (JAva coveraGe faUlt locAlization Ranking)\(^1\) [Ribeiro, 2016] is an open source SFL tool for Java. Jaguar is composed of Jaguar Runner, which collects and generates lists of suspicious program elements, and Jaguar Viewer, which presents the visual information for debugging. Jaguar supports both control-flow and data-flow spectra (i.e., source code lines and definition-use associations, respectively). Figure 5.1 shows the Jaguar’s architecture.

![Figure 5.1: Jaguar’s architecture](image)

Jaguar runner

Jaguar Runner collects control-flow spectrum using the JaCoCo coverage tool\(^2\). To collect data-flow spectra, Jaguar uses the ba-dua coverage tool\(^3\) [de Araujo and Chaim, 2014]. Jaguar invokes JUnit tests of a subject program and collects spectra data for each element covered during the test execution. It iterates over the classes, methods, and then lines or duas.

Jaguar then calculates the suspiciousness score of each element (line or dua) according to a chosen ranking metric. There are ten ranking metrics available for the suspiciousness calculation: DRT, Jaccard, Kulczynski2, McCon, Minus, Ochiai, OP, Tarantula, Wong3, and Zoltar (see Section 2.4.1). Each metric has its range of suspiciousness scores. Jaguar normalizes the scores of all metrics in a range between 0 and 1. Entities with the lowest score are assigned to 0 and entities with the highest score are assigned to 1. Jaguar assigns suspiciousness scores to packages, classes, and methods based on the suspiciousness scores of lines or duas. The score of each package, class, and method is the highest score of their internal lines or duas. This assignment of scores for packages, classes and methods is based on Code Hierarchy technique (see Section 4.2.1).

The Jaguar Runner’s output is a list of lines or duas executed by the tests and their respective suspiciousness scores. A Jaguar’s SFL list is an XML file in which each line or dua is within its code hierarchy structure (package, class, and method). Figure 5.2 shows an example of a Jaguar file with line spectrum\(^4\). Jaguar also provides another XML format, called Flat, which is composed of only lines or duas. Jaguar Runner is available both as an open source plug-in for the Eclipse IDE\(^5\) and a command line tool.

\(^1\)github.com/saeg/jaguar
\(^2\)eclemma.org/jacoco
\(^3\)github.com/saeg/ba-dua
\(^4\)We removed some attributes for better visualization of the report file.
\(^5\)eclipse.org/ide
Figure 5.2: Excerpt of a Jaguar’s XML report file

Jaguar viewer

Jaguar Viewer provides visual information of the SFL lists for debugging within the Eclipse IDE. It colors the suspicious program entities according to their suspiciousness scores. The colors represent four score levels: red (danger) are the most suspicious entities, orange (warning) are those with a high level of suspiciousness, yellow (caution) are those with a moderate suspiciousness level, and green (safety) are the least suspicious ones. The color schema was based on that of the Tarantula tool [Jones et al., 2002]. Additionally, we included the orange color, as done in Gouveia et al. [2013], to increase the distinction among the program elements. The suspiciousness range for the colors is: red ≥ 0.75, 0.75 > orange ≥ 0.5, 0.5 > yellow ≥ 0.25, and 0.25 > green > 0.

Clicking on a program entity in Jaguar leads to the source code on the Eclipse’s editor view. The Jaguar Viewer also has two filtering widgets to facilitate debugging: a text search box to filter by terms of the source code and a slider to narrow down entities by suspiciousness score. Figure 5.3 shows the Jaguar View’s list of methods within the Eclipse IDE. The slider and text search widgets are below the method list.

We created seven views for the Jaguar results. Three views are lists with a single program entity type: methods, lines, or duas. Two views have the complete code hierarchy structure for lines or duas, i.e., a list is ordered by package and its respective classes, methods, and lines or duas. These two views have two frames: one frame for packages, classes, and methods; and another frame that shows lines or duas that belong to the method set in the previous frame. The two other views represent roadmaps (see Section 4.2) and also have two frames: one with the most suspicious methods and another with their respective lines or duas.

Figures 5.4a, 5.4b, 5.4c, 5.5a, 5.5b, 5.6a, and 5.6b show details of the seven Jaguar views, respectively for lines, methods, duas, code hierarchy with lines, code hierarchy with duas, roadmap with lines, and roadmap with duas. In the figures 5.4c, 5.5b, and 5.6b, the Def and Use columns are the line numbers where the definitions and uses occur for a variable in the Var column.

Through these views, Jaguar provides information about the most suspicious entities in a simple visualization scheme, which is adherent to other Eclipse’s views, such as the JUnit and the Package Explorer views. A developer using Jaguar can go straight to the entity s/he wants to inspect without the need of understanding the meaning of a sophisticated graphical model.

Jaguar also allows tracking user actions performed within Eclipse for experimental purposes. The actions tracked by Jaguar include mouse and keyboard events, such as clicking and hovering on the plug-in area, clicking, hovering and code editing in the editor view, adding or removing breakpoints, and testing and debugging execution. Moreover, the Jaguar XML report files can also be used in other visualization tools.
5.2.2 Participants

We selected participants with at least a minimum previous experience (a self-declared basic level) in the Java language. Other non-exclusive criteria were a desirable previous experience in the Eclipse IDE and JUnit. Equivalent technologies, such as TestNG\(^6\), NetBeans IDE\(^7\), and IntelliJ IDEA\(^8\), were also considered as valid experience. The participants must have no previous developing experience in the programs we selected for the experiment (described in the next section) to avoid the impact of previous knowledge on their performance.

The participants’ selection was made by convenience sample [Creswell, 2013]. This is a common practice in user studies in the Software Engineering area due to the difficulty to obtain representative samples.

We invited graduate and undergraduate students from the Institute of Mathematics and Statistics of the University of São Paulo (IME-USP) and the School of Arts, Sciences, and Humanities of the University of São Paulo (EACH-USP). Students from the ACH-2006 — Information Systems Engineering I course at EACH-USP, taught by Professor Marcos L. Chaim in the second semester of 2016, were invited to participate in the experiment. In this case, their participation was considered as a non-mandatory activity of the course.

Additionally, we invited students from the Institute of Mathematics and Computer Sciences of the University of São Paulo (ICMC-USP), the Department of Computer Science of the Federal University of Technology – Paraná (UTFPR), the Department of Computing of the Federal Univer-

---

\(^6\) testng.org
\(^7\) netbeans.org
\(^8\) jetbrains.com/idea
5.2 USER STUDY DESIGN

5.2.3 Subject programs and faults

We selected two open source Java programs for the experiment: *jsoup* and *XStream*. *Jsoup* is an HTML parser composed of 10 KLOC and 468 test cases. *XStream* is a library to (de)serialize objects to/from XML files composed of 17 KLOC and 1,457 test cases.

We set two faults neither too easy nor too hard to find. These faults are not dependent on specific knowledge about the programs’ domain. The *jsoup*’s bug is real, which we found in the program’s repository. The fault is a wrong comparison in an if-statement, which compares two elements using `equals()` instead of `==`. The `XStream`’s bug was seeded by Gouveia et al. [2013] for their experiment. It is a wrong comparison operator of a null condition in an if-statement (`==` instead of `!=`).

We used the Ochiai metric to calculate suspiciousness. We opted for keeping the faulty elements in their original ranking positions. We wanted to verify how the developers inspect the code in a realistic scenario, in which SFL may or may not classify the faulty elements among the first picks.

The faulty lines of *jsoup* and *XStream* were assigned by Ochiai with intermediate scores, 0.65 and 0.67, respectively. The faulty methods of *jsoup* and *XStream* were assigned with a score of

---

9 [github.com/jhy/jsoup](https://github.com/jhy/jsoup)
10 [x-stream.github.io](https://x-stream.github.io)
5.2

(a) List of code hierarchy with lines

(b) List of code hierarchy with duas

Figure 5.5: Jaguar’s lists of code hierarchy with lines and duas

0.65 and 1.0, respectively. In jsoup, the faulty line was ranked in the 18th position, while the faulty method was ranked in the 5th position. For XStream, the faulty line and method were ranked in the 31st and first positions, respectively. This means that both faults are not painted in red—except for the fault in the method list of XStream—in such a way that some false positives may be inspected until reaching the bug. Thus, we are evaluating subjects in which the faulty statements were not so well ranked by Jaguar.

Moreover, these faults are suitable to be found without using the SFL tool. Both faults generate few failing test cases to be inspected, 2 and 5 test cases for jsoup and XStream, respectively. The XStream’s fault throws an exception in which the faulty method is at the top of the stack trace information. The jsoup’s failing test cases call a particular method, which in turn calls the faulty method, leading directly to the faulty method in two steps. These characteristics balance the chances of finding the faults using and not using SFL.

5.2.4 Experimental group design

The study was devised to be performed in two debugging tasks: one task using Jaguar and the other task without using Jaguar. Thus, all participants will have contact with the tool. According to Wainer [2011], this factor avoids a bias effect on the performance of a control group, which may occur when participants use only conventional treatments during experiments. To evaluate RQ10, the participants used only one of the two available SFL lists in the Jaguar task: a list of suspicious lines or a list of suspicious methods.

Two major factors that may impact the results of our experiment are the faults and the order
in which the debugging tasks are performed. The faults may present different difficulty levels of location, resulting in a threat to validity known as *instrumentation* [Campbell and Stanley, 1963]. The order of the tasks may also influence the results: always using the same treatment (task) as the second task may lead to *fatigue*. Moreover, using a conventional technique after using a new one may cause *frustration*, impairing the results of the second task. Both fatigue and frustration are related to a threat to validity known as *maturation* [Campbell and Stanley, 1963].

We defined our groups using the *Latin Square* design [Cochran and Cox, 1957], which avoids the impact of *nuisance factors* in the experiment. Our nuisance factors are the faults used for the experiment and the order in which the tasks are performed. In the Latin Square design, we mitigate the nuisance factors by interchanging their orders through different experimental groups.

An experiment in which each participant performs both the conventional and the proposed approaches would have four experimental groups. However, as we have two versions of Jaguar (lines or methods) and each participant uses only one version, we needed eight experimental groups. Thus, our study has two Latin Square sets: four groups using SFL with a line list (labeled as *JaguarL* and the other four groups using SFL with a method list (labeled as *JaguarM*). The task in which Jaguar is not used was labeled as *Eclipse*. The Jaguar’s line and method lists are those shown in Figures 5.4a and 5.4b, respectively. Table 5.1 details our group design, where the *G* column is the group number.

Thus, our study has the following treatments:

**Jaguar.** In this task, participants can use Jaguar and all other Eclipse resources to search for the bug. It comprises participants that will use either a list of lines (*JaguarL*) or a list of methods (*JaguarM*).
Table 5.1: Experimental groups (G)

<table>
<thead>
<tr>
<th>G</th>
<th>1st task</th>
<th>2nd task</th>
<th>G</th>
<th>1st task</th>
<th>2nd task</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>JaguarM + jsoup</td>
<td>Eclipse + XStream</td>
<td>5</td>
<td>JaguarL + jsoup</td>
<td>Eclipse + XStream</td>
</tr>
<tr>
<td>2</td>
<td>JaguarM + XStream</td>
<td>Eclipse + jsoup</td>
<td>6</td>
<td>JaguarL + XStream</td>
<td>Eclipse + jsoup</td>
</tr>
<tr>
<td>3</td>
<td>Eclipse + jsoup</td>
<td>JaguarM + XStream</td>
<td>7</td>
<td>Eclipse + jsoup</td>
<td>JaguarL + XStream</td>
</tr>
<tr>
<td>4</td>
<td>Eclipse + XStream</td>
<td>JaguarM + jsoup</td>
<td>8</td>
<td>Eclipse + XStream</td>
<td>JaguarL + jsoup</td>
</tr>
</tbody>
</table>

**JaguarL.** It is the Jaguar treatment in which we consider only those that use the list of lines.

**JaguarM.** It is the Jaguar treatment in which we consider only those that use the list of methods.

**Eclipse.** In this task, participants can use all resources of Eclipse to search for the bug, except for Jaguar.

### 5.2.5 Questionnaires

The study has two questionnaires, which were applied before and after the experiment. Both questionnaires are available in Appendix A. The questionnaires do not ask any personal information, ensuring confidentiality and privacy for the participants.

#### Pre-task questionnaire

Before the experiment, we applied an online questionnaire to know the experience level of those who would like to take part in the study. This questionnaire has questions regarding years of experience as professional developers, programming, Java, use of an IDE, and automated testing. We also ask their self-declared skill levels in Java, use of IDE, and automated testing. They must choose one of the four available levels: none, basic, intermediate, and advanced. Feigenspan et al. [2012] claims that self-estimation is a reliable way to measure programming experience. Thus, we used the answers to the self-declared skill levels to balance the distribution through the experimental groups. Those that have at least a basic knowledge level in Java were allocated in one of the experimental groups to perform the experiment (more details on the allocation process on Section 5.2.6).

#### Post-task questionnaire

We built another questionnaire to be filled out by the participants after the debugging tasks. This questionnaire is composed of questions about their performance on the tasks, their opinions and comments about the SFL tool and the performed tasks, the tool’s usability, and to understand their intention to use SFL in the future.

We assessed the behavioral intention to use SFL using the Technology Acceptance Model (TAM) [Davis, 1989]. TAM is a widely known model used to assess how users perceive the usefulness and ease of use of new technologies, which in turn impact on their intention to use such technologies in the future. Perceived usefulness (PU) indicates a person’s opinion on how useful a new technology is. Perceived ease of use (PE) is related to the easiness of using a new technology to perform his/her activities [Davis, 1989]. Both PU and PE are positively correlated with the behavioral intention (BI) of use [Davis, 1989].

We built our TAM questionnaire based on the items presented on Davis and Venkatesh [1996], Brooke [1996], and Feigenspan et al. [2012]. We used a seven-point Likert scale to gather the participants’ degree of agreement with each statement, from (1) totally disagree to (7) totally agree. We included a neutral value since participants may neither agree nor disagree with a statement. The scale items related to the TAM constructs (PU, PE, and BI) are shown in Table 5.2. The italic words are the main concepts addressed by each item.

Regarding the tool’s usability, this questionnaire asks how developers evaluate the main features of the tool, such as the color scheme, filters, and the lists. The questions related to their performance
Table 5.2: Items of TAM

<table>
<thead>
<tr>
<th>Perceived usefulness (PU)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>U1 Using Jaguar <em>improves</em> my performance in the debugging task.</td>
<td></td>
</tr>
<tr>
<td>U2 Using Jaguar in the debugging task increases my <em>productivity</em>.</td>
<td></td>
</tr>
<tr>
<td>U3 Using Jaguar enhances my <em>effectiveness</em> in the debugging task.</td>
<td></td>
</tr>
<tr>
<td>U4 Overall, I find Jaguar <em>useful</em> to perform the debugging task.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Perceived ease of use (PE)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>E5 <em>Learning</em> to use Jaguar is easy for me.</td>
<td></td>
</tr>
<tr>
<td>E6 Interacting with Jaguar does not require a lot of <em>mental effort</em>.</td>
<td></td>
</tr>
<tr>
<td>E7 Overall, I find Jaguar <em>easy to use</em>.</td>
<td></td>
</tr>
<tr>
<td>E8 I find easy to get Jaguar <em>to do what I want</em> it to do.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Behavioral intention of use (BI)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>B9 Assuming I had access to Jaguar, I <em>intend</em> to use it.</td>
<td></td>
</tr>
<tr>
<td>B10 Given that I had access to Jaguar, I <em>predict</em> that I would use it.</td>
<td></td>
</tr>
</tbody>
</table>

ask about where are the faults, what caused them, and how to fix them. We also ask their opinions about the difficulty level of the bugs, suggestions, and comments.

5.2.6 Participants’ allocation

We used the *Haphazard Intentional Allocation* (HIA) method [Fossaluza et al., 2015] to distribute the participants through the experimental groups. The HIA method uses the *Aitchison compositional distance* [Aitchison, 1983] to calculate in which group a new participant must be allocated considering desirable features of the experiment and the weights (ω) associated with them. A perturbation parameter (ϵ) is used to add some randomness factor to the deterministic process.

This allocation method allowed us to balance participants with different technical skill levels through the groups, even for a small number of participants. We used three skill features in our study: knowledge levels in Java, IDEs, and automated testing, which are gathered in the pre-task questionnaire.

To determine our ϵ and ω, we created a randomized population of 2,000 individuals. We ran HIA for different samples sizes from this initial population, from 10 to 100, using different values for ϵ (from 0.01 to 0.35) and ω (e.g., 1-1-1, 2-1-1, 5-1-1, for Java, IDEs, and testing, respectively). Each combination was run four times. Then, we chose the smallest ϵ and ω that better allocated the samples. We carried out this procedure because we did not know *a priori* the number of participants that would take part in the study. Thus, the weights of our features were ω = 2 for Java, ω = 1 for IDEs, and ω = 1 for automated testing. We set a value of 0.05 for ϵ in our study.

5.2.7 Experimental procedure

The experiment was carried out remotely using virtual machines (VMs). The VMs ensure that the participants have the same environment to perform the experiments. These VMs contain all resources required for the experiment: a *Lubuntu 14.04* operational system; Eclipse IDE with the Jaguar plug-in installed; the subject programs and faults; training material on Jaguar, and the instructions for the experiment. We created two VMs, one for Jaguar with lines and the other for Jaguar with methods. Each VM has four users, one for each experimental group. The participants needed a computer with 2GB RAM and the VirtualBox program to run the VM. Thus, they could perform the experiment from anywhere. We defined a deadline of three weeks from the disclosure of the experiment to finish it.

Figure 5.7 depicts the procedure of our user study. We invited students through an e-mail consent letter. Those students that wanted to participate in the experiment filled out the pre-
task questionnaire. We implemented the HIA method on a web form to automatically allocate the participants. Those with at least a basic knowledge of Java were allocated in the experimental groups, receiving an ID, a user login, a password, and a link for downloading the VM. The ID was provided to keep the participants’ anonymity. This ID is used to perform the debugging tasks and to fill out the final questionnaire, which allowed us to link the log files to the questionnaires of each participant.

The participants read the training material and the instructions for the experiment. The training material is composed of a document and a video explaining how to use the SFL tool, and a tryout version of the tool to allow the participants to interact with it before starting the tasks. The participants were recommended to spend up to 30 minutes for each task. However, we let them control the time by themselves.

To start the first task, the participants need to click on a start button, which allows Jaguar to create the log file. To finish the second task, they should click on a stop button to send the log file to our server. To make sure that the log files are sent, we created a script to send the log files whenever the user session is logged off or during the VM’s shutdown.

After finishing the tasks, the participants were asked to respond the post-task questionnaire, which was available on a web form. We then used the data collected in the questionnaires and in the log files to perform our analysis. The training document, instructions for the experiment, and consent letter are available in Appendix A.

![Figure 5.7: Experimental procedure](image)

### 5.3 Results and discussion

This section presents the user study results, guided by our research questions (see Section 5.1). Before going to the research questions, we assessed how balanced is the distribution of participants through the groups regarding our study’s nuisance factors: the faults and the order of the tasks. By doing so, we can verify if the comparison of our results for those using the SFL tool (Jaguar) and without using it (Eclipse) has a fair distribution of the faults and the order in which the tasks were performed.

Then, we present the answers to our research questions. We also present the results of other concerns regarding the impact of the developers’ experience and their behavior during the tasks.
5.3.1 Participants’ distribution

Figure 5.8 shows how the participants were distributed through the groups using the haphazard intentional allocation (HIA) method. In total, 41 participants started the experiment, while 31 of them finished it. Ten people filled up the pre-task questionnaire but did not perform the experiment, as shown by the gray bars. We excluded five other participants due to incomplete data issues, such as the absence of a questionnaire or log file, as shown by the red bars. Thus, our experiment had 26 valid participants, shown by the blue bars. Nielsen [2012] recommends at least 20 users for usability studies that aim to apply statistical tests, which indicates that our sample suffices for conducting a statistical analysis. To keep the participants’ anonymity, we did not ask for personal information. Thus, we do not know what institutions they came from.

Figure 5.8 shows that the HIA method was successful in allocating at least 4 participants for each group, keeping a balanced distribution for the participants of our study. However, some groups had few valid participants because some of them filled out the pre-task questionnaire but did not perform the whole study. Thus, we assessed whether the faults and order of the tasks have significant impacts on the results. Figure 5.9 shows a Venn diagram describing how many participants used Jaguar to inspect the jsoup and XStream programs and in which order Jaguar was used. Tables 5.3 and 5.4 summarizes how many participants used Jaguar, Eclipse, JaguarL, and JaguarM for each fault and in which order the tasks were performed, respectively.

![Figure 5.8: Participants’ allocation](image)

<table>
<thead>
<tr>
<th>Group</th>
<th>Participants</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>5</td>
</tr>
</tbody>
</table>

valid (26) excluded (5) undone (10)

Table 5.3: Distribution of participants through the faults

<table>
<thead>
<tr>
<th>Fault</th>
<th>Jaguar</th>
<th>%</th>
<th>Eclipse</th>
<th>%</th>
<th>JaguarL</th>
<th>%</th>
<th>JaguarM</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>jsoup</td>
<td>15</td>
<td>57.69</td>
<td>11</td>
<td>42.31</td>
<td>6</td>
<td>23.08</td>
<td>9</td>
<td>34.61</td>
</tr>
<tr>
<td>XStream</td>
<td>11</td>
<td>42.31</td>
<td>15</td>
<td>57.69</td>
<td>6</td>
<td>23.08</td>
<td>5</td>
<td>19.23</td>
</tr>
</tbody>
</table>

We used the full bayesian significance test (FBST) [Bernardo et al., 2012] to evaluate the distribution of participants through the groups. The FBST is used to assess distributions of samples in contingency tables (crosstabs) [Pereira and Stern, 1999]. FBST uses Monte Carlo simulation to estimate a normal distribution based on our sample. Also, this test can be applied to small samples to empirically estimate if the distribution is normal regarding a standard deviation.

11Whenever we refer to Jaguar or SFL, we are considering both JaguarL and JaguarM.
5.3.2 RQ6: Can SFL help more developers to locate faults?

SFL is more effective with statistical significance. Moreover, the difference in effectiveness between Jaguar and Eclipse has a medium effect size. Table 5.5 shows the statistical test results for RQ6 comparing the results of Jaguar, JaguarL, and JaguarM to Eclipse. Our null hypothesis (H₀) is that the use of SFL is equal or less effective than the use of Eclipse without SFL. Bold values represent cases in which the null hypothesis is refuted, i.e., SFL is more effective with statistical significance.

We applied FBST to assess the effectiveness of the techniques. We measured the effect size using odds ratio [Chen et al., 2010], which is suitable for our binary results of effectiveness (found or not found). The effect sizes in Table 5.5 are insignificant (I), small (S), medium (M), or large (L). According to Chen et al. [2010], the effect size (ES) scale for odds ratio is insignificant for ES < 1.68, small for 1.68 ≤ ES < 3.47, medium for 3.48 ≤ ES < 6.71, and large for ES ≥ 6.71.

Table 5.5: Statistical results for RQ6

<table>
<thead>
<tr>
<th>H₀</th>
<th>P-value (%)</th>
<th>Effect size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jaguar ≤ Eclipse</td>
<td>1.66</td>
<td>3.6 (M)</td>
</tr>
<tr>
<td>JaguarL ≤ Eclipse</td>
<td>4.65</td>
<td>3.0 (S)</td>
</tr>
<tr>
<td>JaguarM ≤ Eclipse</td>
<td>11.29</td>
<td>4.2 (M)</td>
</tr>
</tbody>
</table>

The use of JaguarL also resulted in better effectiveness with statistical significance and small
effect size. JaguarM did not result in better effectiveness with statistical support.

Table 5.6 shows the number of participants who found faults using Jaguar, Eclipse, JaguarL, and JaguarM. The Total column represents the number of successful developers for both projects. This table also shows how many participants found faults by project. The percentages are relative to the number of participants with similar settings. For example, 4 of 5 developers using JaguarM in XStream found the bug.

Table 5.6: Effectiveness of participants by technique and project

<table>
<thead>
<tr>
<th>Technique</th>
<th>Total</th>
<th>% jsoup</th>
<th>% XStream</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jaguar</td>
<td>12</td>
<td>46.15</td>
<td>7</td>
<td>63.63</td>
</tr>
<tr>
<td>Eclipse</td>
<td>5</td>
<td>19.23</td>
<td>3</td>
<td>20.00</td>
</tr>
<tr>
<td>JaguarL</td>
<td>5</td>
<td>41.67</td>
<td>3</td>
<td>50.00</td>
</tr>
<tr>
<td>JaguarM</td>
<td>7</td>
<td>50.00</td>
<td>4</td>
<td>80.00</td>
</tr>
</tbody>
</table>

Almost half of the participants were successful using SFL. On the other hand, using Eclipse only, less than one-fifth of them found a bug. Four participants found the faults using both Jaguar and Eclipse, which means that 12 of 13 participants who found bugs in our experiment were successfully using Jaguar. Conversely, only one participant found a fault using Eclipse and did not find the other one using Jaguar. These results show that developers can locate more faults when using an SFL tool.

Nearly half of the participants found the bugs using SFL. We believe that factors such as experience level, reduced time for the debugging tasks, and lack of knowledge about the code led to this modest success rate.

The effectiveness results obtained in our study corroborates previous studies, in which SFL improved fault localization [Gouveia et al., 2013; Xia et al., 2016; Xie et al., 2016]. In the study of Parnin and Orso [2011], only more experienced developers took advantage of using SFL.

5.3.3 RQ7: Can SFL help developers to locate faults faster?

There are no significant differences in efficiency using SFL. As we measure efficiency by the time spent for those who found bugs, our sample size is more limited to conclude using statistical tests. The values for the statistical tests are shown in Table 5.7. Our null hypothesis (H₀) is that the use of SFL is equal or less efficient than the use of Eclipse without SFL. We used the Wilcoxon Rank-Sum test [Wilcoxon, 1945] for the hypothesis tests and Cliff’s delta [Cliff, 1993] to measure the effect size. These statistical methods are suitable for the efficiency data of our study, which is composed of non-parametric numerical values. The values for Cliff’s delta range from -1 to 1. According to Macbeth et al. [2011], the effect size (ES) scale for Cliff’s delta is insignificant (I) for -0.2 ≤ ES ≤ 0.2, small (S) for 0.21 ≤ ES ≤ 0.5 and -0.5 ≤ ES ≤ -0.21, medium (M) for 0.51 ≤ ES ≤ 0.8 and -0.8 ≤ ES ≤ -0.51, and large (L) for ES > 0.8 and ES < -0.8.

Figure 5.10 presents box plots for the time spent in minutes by the developers to locate the bugs using Jaguar and Eclipse. The median of time spent using Jaguar is lower than using Eclipse, respectively 16.55 and 23 minutes, although the times using Jaguar vary more than using Eclipse. These results indicate that Jaguar can reduce the time spent to locate bugs.

Although the time is not improved with statistical significance as in Xia et al. [2016], SFL did not impair the efficiency as occurred in Xie et al. [2016]. We believe that, due to the size of our programs, developers can cut off time in debugging while using SFL, since there are a large number of classes and methods to inspect. An SFL tool that can classify well faulty components can reduce the search space and, thus, improve the debugging efficiency.
5.3.4 RQ8: Can SFL guide more developers to inspect faulty code?

In this question, we want to know if SFL leads developers closer to the faulty regions, which may increase their chances of finding bugs. To assess RQ8, we verify whether SFL can help developers to inspect the faulty line and to locate the faulty method.

Inspection of the faulty line

**SFL leads developers closer to the faulty lines.** First, we evaluated if there is some relationship between clicking on the faulty line at least once and finding the faults. Since each participant performed two tasks, we built a crosstab (see Table 5.8) for this comparison considering the results of both tasks. The Yes and No lines are the numbers of participants who clicked or not clicked in the faulty line at least once, respectively. The Yes and No columns are the debugging result: found or not found. Using FBST, we obtained a *p*-value of 0.3%, and an odds ratio value of 6.1, which shows that there is a relationship between clicking on the faulty line and finding the bug with statistical significance and medium effect size.

Based on this result, we verified whether the SFL tool leads more participants to click at least once on the faulty lines. The crosstab comparing those who inspected the faulty line using Jaguar and Eclipse is shown in Table 5.9, where Yes and No lines have the number of participants who inspected or not inspected the faulty line at least once using Jaguar, respectively. The same rationale is applied to the Yes and No columns for those using Eclipse. We measured statistical significance and effect size using FBST and odds ratio, respectively. Our null hypothesis (H0) for the inspection of lines is that SFL leads to an equal or small number of developers that inspect the faulty line compared to Eclipse.

The results show that there is a relationship between the use Jaguar and inspecting the faulty line with statistical significance and medium effect size, as shown in Table 5.10 (Table 5.10 follows the same structure of Table 5.5). We also performed this comparison for JaguarL and Eclipse.

<table>
<thead>
<tr>
<th></th>
<th>H0</th>
<th>P-value (%)</th>
<th>Effect size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jaguar ≤ Eclipse</td>
<td>79.9</td>
<td>0.1 (S)</td>
<td></td>
</tr>
<tr>
<td>JaguarL ≤ Eclipse</td>
<td>38.1</td>
<td>0.6 (M)</td>
<td></td>
</tr>
<tr>
<td>JaguarM ≤ Eclipse</td>
<td>26.67</td>
<td>-0.52 (M)</td>
<td></td>
</tr>
</tbody>
</table>
5.3 RESULTS AND DISCUSSION

Table 5.8: Crosstab for clicking on the faulty line at least once and debugging result

<table>
<thead>
<tr>
<th>Clicked</th>
<th>Found</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>No</td>
<td>27</td>
<td>0</td>
</tr>
<tr>
<td>Yes</td>
<td>8</td>
<td>17</td>
</tr>
</tbody>
</table>

Table 5.9: Crosstab of those who clicked on the faulty line at least once using Jaguar and Eclipse

<table>
<thead>
<tr>
<th></th>
<th>Jaguar</th>
<th>Eclipse</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>No</td>
<td>7</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Yes</td>
<td>12</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

and for JaguarM and Eclipse. JaguarM also lead more developers to inspect the faulty line with statistical significance and large effect size. JaguarL did not reach statistical significance.

Location of the faulty method

The SFL tool leads more developers to identify the faulty method. We compared the participants who correctly answered the faulty method for both debugging tasks in the post-task questionnaire. Table 5.11 shows the number of participants who found the faulty methods using Jaguar, Eclipse, JaguarL, and JaguarM, where the Total column shows the total for both projects. The percentages are relative to the number of participants with similar settings.

For this comparison, our null hypothesis ($H_0$) is that SFL leads to an equal or small number of developers that locate the faulty method compared to Eclipse without SFL. We used FBST and odds ratio for the statistical analysis. As shown in Table 5.10, developers using SFL were more effective to locate the faulty method with statistical significance. The differences between Jaguar and Eclipse presented a medium effect size. JaguarL also reached statistical significance with large effect size.

Our analysis indicates that SFL improves the interaction with the faulty lines, which may increase the chance of finding bugs. Clicking on the faulty line does not guarantee to find a bug [Parnin and Orso, 2011]. However, it is necessary to at least reaching a faulty code excerpt to understand and fix it.

SFL was also effective to help developers to identify at least the faulty method. A method is the program entity that often contains complete excerpts of program’s functionalities. It is also closest to the faulty lines. Thus, identify a faulty method may be the first step to understand a fault. Indeed, in the survey conducted by Kochhar et al. [2016], developers deemed method as the preferred code level that fault localization techniques should provide. Thus, a tool that guides developers closer to bugs can play an important role in the debugging process.

Table 5.10: Statistical results for RQ8

<table>
<thead>
<tr>
<th>$H_0$</th>
<th>RQ8 – Faulty line</th>
<th>RQ8 – Faulty method</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>P-value (%)</td>
<td>Effect Size</td>
</tr>
<tr>
<td>Jaguar $\leq$ Eclipse</td>
<td>0.33</td>
<td>6.11 (M)</td>
</tr>
<tr>
<td>JaguarL $\leq$ Eclipse</td>
<td>8.64</td>
<td>4.0 (M)</td>
</tr>
<tr>
<td>JaguarM $\leq$ Eclipse</td>
<td>1.33</td>
<td>9.16 (L)</td>
</tr>
</tbody>
</table>
Table 5.11: Participants who found the faulty methods

<table>
<thead>
<tr>
<th>Technique</th>
<th>Total</th>
<th>% jsoup</th>
<th>% XStream</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jaguar</td>
<td>18</td>
<td>69.23</td>
<td>9</td>
<td>81.82</td>
</tr>
<tr>
<td>Eclipse</td>
<td>8</td>
<td>30.77</td>
<td>3</td>
<td>27.27</td>
</tr>
<tr>
<td>JaguarL</td>
<td>8</td>
<td>66.67</td>
<td>3</td>
<td>50.00</td>
</tr>
<tr>
<td>JaguarM</td>
<td>10</td>
<td>71.43</td>
<td>6</td>
<td>66.67</td>
</tr>
</tbody>
</table>

5.3.5 RQ9: Are the study participants intended to use SFL in practice?

Most participants showed the intention to use Jaguar in the future. We assessed the tool’s perceived usefulness (PU), perceived ease of use (PE), and behavioral intention of use (BI), adapting the Technology Acceptance Model (TAM) to the SFL domain. The items of our TAM questionnaire are in Table 5.2.

First, we verified the internal validity of our questionnaire regarding the TAM items in producing valid results. We evaluated the reliability of our questionnaire using the Cronbach’s Alpha [Carmines and Zeller, 1979], which assesses whether a set of items of a test is closely related. We obtained alpha values of 0.95 and 0.97 for items of perceived usefulness and perceived ease of use, respectively. These values are higher than the threshold of 0.8 [Carmines and Zeller, 1979], which indicates that our questionnaire is reliable.

We also measured the factorial validity, which aims to verify whether the items of PU and PE form distinct constructs [Laitenberger and Dreyer, 1998]. Items from the same construct tend to be more correlated. A threshold to indicate correlated items is 0.7 [Jae-On Kim, 1978]. Table 5.12 shows the correlation values for the items of PU (U1 to U4) and PE (E1 to E4), where bold values are those above the threshold. The results show that PU is highly related to the items designed for this purpose, except for U4 (usefulness), which had a value slightly lower than the threshold. PE is also highly related to the items E1 to E4.

Table 5.12: Factorial analysis of the TAM items

<table>
<thead>
<tr>
<th>PE</th>
<th>PU</th>
<th>PE</th>
<th>PU</th>
</tr>
</thead>
<tbody>
<tr>
<td>U1</td>
<td>0.28</td>
<td>0.94</td>
<td>E5</td>
</tr>
<tr>
<td>U2</td>
<td>0.38</td>
<td>0.84</td>
<td>E6</td>
</tr>
<tr>
<td>U3</td>
<td>0.24</td>
<td>0.84</td>
<td>E7</td>
</tr>
<tr>
<td>U4</td>
<td>0.61</td>
<td>0.66</td>
<td>E8</td>
</tr>
</tbody>
</table>

To ensure that the responses regarding behavioral intention to use are accurate, we assessed the correlation between the three TAM constructs using the Pearson’s product correlation coefficient (r) [Cohen, 1977]. The r values vary from -1 to 1, where -1 means a negative correlation, 0 means no correlation, and 1 means a positive correlation. The results are shown in Table 5.13, which indicates that both usefulness and ease of use are positively correlated with behavioral intention of use. The p-values for each comparison indicate that the correlations are statistically significant.

Table 5.13: Correlation among TAM constructs

<table>
<thead>
<tr>
<th>PU x BI</th>
<th>PE x BI</th>
<th>PU x PE</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>0.87</td>
<td>0.61</td>
</tr>
<tr>
<td>P-value (%)</td>
<td>9.22x10^{-7}</td>
<td>8.77x10^{-2}</td>
</tr>
</tbody>
</table>

Thus, our questionnaire and the TAM constructs are reliable to be used for assessing the participants’ answers about the Jaguar tool.
Perceived usefulness

Most participants deemed the SFL tool useful. The aggregate rating scale for the PU items ranges from 4 to 28. The values assigned by the participants had a median of 22, an average of 20.65, and a mode of 26.

Figure 5.11 shows the answers for usefulness for each item. The number of participants that totally agree, agree, or somehow agree varied from 15 (57.7%) for U3 (effectiveness) to 21 (80.7%) for U4 (usefulness). The maximum disagreement level was 7 (26.9%) for U3. The U3 item is directly related to task success, which may explain why this item was worst rated.

Perceived ease of use

The participants rated the SFL tool as easy to use. The answers for the aggregated items had a median of 24, an average of 22.46, and a mode of 24. The rating scale ranges between 4 and 28.

Figure 5.12 shows the answers for each PE item. Participants that assigned some level of positive agreement varied between 20 (76.9%) for E8 (task easiness) and 21 (80.8%) for E5 (learning), E6 (mental effort), and E7 (ease of use). Only 3 (11.5%) participants assigned some disagreement level for each PE item.

Behavioral intention of use

Most participants showed interest in using the SFL tool in the future. Their answers for the BI items had a median of 13, an average of 11.5, and a mode of 14. The rating scale ranges between 2 and 14.

Figure 5.13 shows the answers regarding behavioral intention of use. 18 (69.2%) and 19 (73.1%) participants assigned some positive agreement level for BI9 (intention of use) and B10 (prediction), respectively. These results indicate that they are willing to use an SFL tool for their debugging activities if such a tool is available.

The SFL tool received positive ratings even from those who did not find bugs since the average of positive agreements with the TAM questionnaire is always higher than the number of developers who successfully found the bugs. This fact indicates that the participants of our study
are willing to use techniques to support debugging even when they were not successful on a first try. Thus, we believe that techniques able to highlight faulty program elements in most cases will be well accepted for practical use. Despite the need for improvements, SFL tools should be widely disseminated to increase the likelihood of being adopted in real settings.

5.3.6 RQ10: The level of code information impacts on the fault localization performance?

The level of code information (suspicious methods or lines) did not present differences regarding effectiveness or efficiency. Table 5.14 shows the statistical results for RQ10 comparing JaguarL and JaguarM. The null hypotheses are that the use of JaguarL or JaguarM has the same effectiveness and efficiency. As in RQ6, we used FBST to test the null hypothesis and odds ratio to measure the effect size of effectiveness. For the efficiency evaluation, we used the Wilcoxon Rank-Sum test and Cliff’s delta, as in RQ7.

Table 5.14: Statistical results for RQ10

<table>
<thead>
<tr>
<th></th>
<th>Effectiveness</th>
<th></th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>$H_0$ JaguarL = JaguarM</td>
<td>P-value (%)</td>
<td>Effect size</td>
<td>P-value (%)</td>
</tr>
<tr>
<td></td>
<td>67.77</td>
<td>0.71 (I)</td>
<td>75.51</td>
</tr>
</tbody>
</table>

In both cases, the null hypothesis was confirmed. The p-values indicate that there were no differences when comparing JL and JM for both effectiveness and efficiency. The effect sizes are insignificant and small, respectively. Thus, SFL with information of method or lines were more effective than debugging without using SFL, but they presented similar performance between them.
5.3.7 Programming experience

Table 5.15 shows the participants’ years of experience in Java, professional programming, use of IDEs, and automated testing. The No column shows those who have no experience, while the 0 column shows those who have less than one year of experience. The 10 column shows those who have 10 or more years of experience. Few participants have large experience. Regarding Java, only 3 of them have more than 5 years of experience, while 16 have from 3 to 5 years. Most participants have at least a minimum experience with professional programming. Also, almost all participants have little or no experience in automated testing.

<table>
<thead>
<tr>
<th>Experience</th>
<th>No</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>10</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Professional</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>IDE</td>
<td>0</td>
<td>2</td>
<td>6</td>
<td>7</td>
<td>5</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Testing</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Programming</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>7</td>
<td>6</td>
<td>6</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
</tbody>
</table>

Most developers who found a bug using Jaguar have 2 or 3 years of experience in Java. Two of three developers with at least 10 years of experiment found bugs using SFL, and one of them did not find the other bug without using SFL. From those who found at least one bug, 4 have no previous experience with automated testing, and 7 have at most 1 year. Regarding professional experience, only one developer with no experience found a bug, three have less than one year, and five of them have between 1 and 2 years as developers.

Even developers with little experience can take advantage of SFL to locate bugs. Since most of our sample is composed of less experienced developers, the effectiveness results indicate that SFL can help them to locate bugs. The tool is also useful to guide towards the faulty code. The overall low level of experience, especially in automated testing, may help to explain why only half participants found a bug. However, the performance was significantly better for those using SFL, as shown in Section 5.3.2.

It is important to note that the debugging tasks performed in this study used real programs and the developers had no previous knowledge of the code. Moreover, we kept the faulty line and methods in the positions assigned by the Ochiai ranking metric. Thus, the lines were ranked in intermediate positions, as we reported in Section 5.2.3.
5.3.8 Developers’ comments and observations

In the post-task questionnaire, the participants answered several questions regarding the SFL tool, about its resources, suggestions for improvement, and their opinions about the experiment settings. We also included in this section some observations from the log files.

When asked about how the tool helped to locate the bug, some participants answered that the tool put them closer to the faulty region, helping them to identify the fault. Others participants reported that even not finding the fault, the tool helped at least to indicate the faulty class and method. Indeed, some developers indicated the right method and class in the questionnaire as we discussed in the previous section. For example, three participants described how the tool helped her/him to locate the bug:

“Pinpointing what most likely is the problem when starting to debug a code totally unknown was very useful. [...] without the tool I got lost trying to understand all that could be behind the problem [fault] [...]”

“It [Jaguar] gives a direction on where to start by listing the highest chances of fault.”

“[Jaguar] helped only to focus on the code area where the problem was.”

Regarding the SFL resources, most participants considered that suspiciousness lists, highlights on the faulty code, color scheme, and suspiciousness scores are the most useful information provided by Jaguar. Few participants used the text search and the slider filters. Several participants suggested improvements for Jaguar. Some of them would like to see why the tool deemed a line as suspicious. Other frequent suggestion was including information about suspicious variables. The developers also want to see information about the relationship between suspicious elements and failed tests. Another feature they want to see is the suggestion of potential fixes.

Some participants complained about the limited time to perform the tasks, while others commented on the difficulty of debugging without previous knowledge of the code. Indeed, such factors may have impaired the performance of some participants.

From these observations, we conclude that SFL tools are useful to guide developers to suspicious code excerpts. However, such tools should be integrated with other existing development resources to help developers to understand potential reasons that cause faults.

5.3.9 Developers’ behavior

We observed how the participants interacted with Jaguar and without it in the experiment. In what follows, we describe the main findings regarding their behavior during the debugging tasks.

Use of SFL during the tasks

Most participants used Jaguar throughout the task. To evaluate this issue, we divided the task time performed by each participant into three parts to assess how long they interacted with the Jaguar during their tasks. From the 26 participants, 24 used Jaguar in the first part, 20 of them in the second part, and 21 used in the third part. Thus, most participants did not give up on using SFL, even when the fault was not found. This result is opposite to that shown by Parnin and Orso [2011], in which developers would give up on using SFL in the presence of false positives. None of the participants of our study reported having given up on Jaguar, even those who answered that the tool did not help them. Only two participants that did not find the bug in the SFL task gave up on the task in a short time, both after spending around 15 minutes.

Contrary to the results of Xie et al. [2016], most developers in our experiment inspected the SFL list before inspecting the source code. As the programs used in Xie et al. [2016] are small, some of them with a single file, it seems reasonable to look at the code before (first scan) using an SFL tool. However, the programs we used are large, with hundreds of class files. Thus, such first scan pattern
may not hold when debugging more complex programs. The behavior pattern of the developers in our experiment was somehow similar to that observed in the study of Xia et al. [2016]. In general, the developers (1) clicked on some suspicious lines/methods in the SFL lists, (2) navigated through the source code and then (3) looked at the test cases. This process was repeated by most of them until finding the bug or the time ran out. Several participants changed the code and ran the tests aiming at fixing the faults.

Most developers used both SFL and JUnit together during the SFL task. We observed that some of them switched between SFL and JUnit several times. Such fact suggests that relating suspicious elements to their respective failed tests, as suggested by some participants (and also suggested by Parnin and Orso [2011]) may help to understand possible causes of a fault. Few developers used the Eclipse’s debugging resources (i.e., JDT Debug) during the tasks. Only 5 participants used breakpoints and the Eclipse Debugger in the SFL task, and 6 of them used in the other task.

Inspection of the SFL lists

In general, developers inspected only the most suspicious elements in the SFL lists. These most suspicious ones were inspected several times during the tasks. The number of distinct elements inspected by the participants has a median value of 6. 16 participants (61.5%) inspected at most 7 distinct elements, and 23 (88.5%) inspected at most 20 distinct ones. Regarding the suspiciousness scores, 13 (50%) participants inspected at most elements within the two highest scores. 21 (80.1%) participants inspected at most elements with six different scores. The median for distinct scores is 3. This fact suggests that the value of two highest scores that we chose for our filtering strategy Level Score (see Section 4.4.2) is close to the behavior observed in practice.

The developers did not inspect the elements in the order proposed by the SFL lists. 21 (80.1%) of them started the task by inspecting the top 1 element, but only 3 (11.5%) followed the list in order until the fourth element. Most developers inspected the high ranked elements several times. The average for the most inspected element by participant is 9.61, with a median of 6. We did not observe differences in the navigation between the SFL lists of lines and methods.

The results presented in the TAM analysis suggest that developers are willing to spend time using an SFL tool for their debugging activities. However, lists with a great number of program elements are useless, since they inspect at most 7 distinct ones in most cases, and no more than 20 elements for almost all participants. Our results show that most participants interact with Jaguar by clicking on the suspicious elements, and then by navigating through the source code around them. Thus, the suspicious elements act as starting points to understand and reach the faults. Even inspecting few suspicious elements, SFL helped more participants to find the bugs, to find the faulty methods, and to inspect the faulty lines. This fact indicates that SFL may be useful even when the faulty line is not ranked among the top ones.

Moreover, the order in which the elements are presented does not seem to be important. Thus, other ways to present the most suspicious elements can be explored, for example, presenting elements that are more closely related together, suspicious variables, small subsequences, and so on.

5.3.10 Factors in favor of the practical use of SFL

Throughout our research questions, we identified some factors that suggest that SFL can be used practice:

(1) SFL improved the effectiveness to locate faults, even when a fault is not ranked among the top ones. Moreover, even less experienced developers were able to find bugs using SFL. The efficiency was not significantly better for SFL; the time to complete the tasks using Jaguar was just slightly shorter.

(2) There is a correlation between clicking on a faulty line and finding its bug. Although the perfect bug understanding is not accomplished (as shown in previous studies [Parnin and Orso, 2011; Xia et al., 2016; Xie et al., 2016]), reaching the faulty site helps to locate bugs.
3) SFL led more participants to inspect the faulty line, and also led developers to locate the faulty methods, even for those who did not find the bug. Thus, SFL can be useful to provide hints for a developer investigating possible causes of faulty behaviors.

4) Participants deemed Jaguar as easy to use and useful. Moreover, they showed the intention to use it in the future, including those that did not find the bug. We believe that most developers in the software industry do not know about SFL techniques, and disseminating them is fundamental for its adoption. Also, researchers should propose new ways to report suspicious results to provide techniques more suitable for practical use.

5.3.11 Factors against the practical use of SFL

By observing the navigation logs and from the participants’ comments and suggestions, we identified factors that impair the adoption of SFL:

1) Long lists of suspiciousness are useless, and may confuse developers. Most developers did not inspect elements below the top-20. Thus, excessive information should be avoided by SFL tools.

2) Developers missed information explaining why elements are deemed as suspicious. Also, several participants would like to see additional information beyond the SFL lists, such as variables and test cases related to suspicious elements, which means that they need more information to understand the bugs. Thus, these developers’ observations suggest that strategies to add contextual information for SFL can be useful in practice.

3) Most participants in our study have little or no experience with automated testing, which is not an uncommon fact in real settings. Indeed, several software companies do not make use of automated testing or make limited use of it. This fact limits the possibilities of adoption of SFL techniques in such scenarios.

5.4 Threats to validity

The internal threats to validity are the Jaguar tool and the SFL lists it provides. We opted to use single lists of faulty methods and lines that are similar to other Eclipse’s views. Our objective in this study is not to evaluate the Jaguar tool by itself, but a general SFL tool. However, the use of other visualization schemes may lead to different results.

A threat to the construct validity is the time to perform the tasks. We set a time limit of 30 minutes for each task, which may impact on the developers’ performance, especially for the less experienced developers. However, the time is the same for both tasks. A long period for the tasks can also impact on the performance since some participants would get tired in the second activity.

All participants did not have previous knowledge about the programs used in our experiment. This scenario avoids the impact of previous knowledge in the debugging performance. However, a scenario in which developers have previous knowledge about the code may lead to different results.

Another threat to the construct validity is the position of the faults in the SFL lists. We opted for keeping the faulty lines and methods in the original positions where they were assigned by Ochiai. We intended to assess a realistic scenario that can occur using an SFL tool. We also ran the faulty programs using Tarantula and Jaccard to avoid the influence of a ranking metric in the results. Both metrics assigned the same positions for both faults. The faulty methods were ranked in better positions than the faulty lines, which may have impaired the participants’ performance.

Regarding the external threats, we chose two faults neither too easy nor too hard to find. Both faults are independent of knowledge about the program domain. However, different faults may lead to different results. We also used the Latin square design to distribute the faults across the techniques and the order in which they are performed to avoid biases.

A final threat to the external validity is related to the participants of the study. They are undergraduate and graduate students, which may not generalize to a representative sample of the population of developers. Most of them have some professional experience, and few of them have
large professional experience. We used the HIA method to keep the distribution of developers balanced through the experimental groups.

5.5 Final remarks

This chapter presented our user study that aimed to understand how developers use SFL and to what extent SFL can be useful in practice. The study was carried out with 26 developers who performed two debugging tasks using two faults from two real programs. We investigated several factors, some of them were already investigated (e.g., effectiveness) and others factors not previously assessed to the best of our knowledge (e.g., guidance to lead developers towards faulty excerpts, user acceptance).

We found that SFL improved the developers’ fault localization ability, leading more developers to locate faults. However, it did not reduce the time spent on the tasks.

We also found that there is a relationship between inspecting the faulty line and locate the bugs. Thus, finding the code region where faults occur may be the first step to locate bugs. SFL leads more participants to inspect the faulty lines and to locate the faulty methods, which may increase their chances to find bugs.

The participants showed intention to use SFL in the future, deeming the SFL tool (Jaguar) useful and easy to use. Also, they would like to see more information related to the suspicious elements, such as variables, test cases, and explanations about faulty behaviors.

Analyzing their interaction with Jaguar, we found that they navigate more frequently among the well-ranked elements of the SFL lists. Thus, to develop tools for practical use, it is necessary to pinpoint faulty elements among the first picks, as already observed in previous work [Parnin and Orso, 2011; Xia et al., 2016]. Despite the need for improvements in SFL, we believe that it is necessary to create and publicize SFL tools in the software communities to attract the interest of practitioners.

As found in previous work [Parnin and Orso, 2011; Xia et al., 2016; Xie et al., 2016], the perfect bug understanding does not hold, although we found that clicking on the faulty site is correlated to find its bug. Contrary to Parnin and Orso [2011], in this experiment less experienced developers also took advantage of SFL to locate the bugs. In our experiment, SFL is more effective, but it is not more efficient, as in Xie et al. [2016].

Debugging is a strictly intellectual activity, which often requires a lot of mental effort. Thus, SFL techniques should not try to replace the developer’s role. Instead, they should provide more precise and useful information to support developers to locate bugs.

In the next chapter, we present a study that assessed the use of data-flow spectrum in fault localization.
Chapter 6

Use of data-flow in spectrum-based fault localization

In this chapter, we investigate whether data-flow spectrum improves SFL. Most SFL techniques use only control-flow spectrum, such as statements or branches. Data-flow spectrum provides more information and explores different relationships among program entities during test execution, which may improve fault localization.

In this study, we compare the performance of definition-use associations (duas) and line spectra to classify faults and their overhead impacts on real open source projects. This study was conducted along with Henrique Lemos Ribeiro, who concluded his Master’s degree [Ribeiro, 2016] on this topic advised by Professor Marcos L. Chaim. In what follows, we describe our methodology to compare data- and control-flow for SFL, results, discussion, threats to validity, and our final remarks.

6.1 Research questions

This study was designed to investigate the following question: Can data-flow spectrum improve SFL? We assess the effectiveness and efficiency of data-flow comparing it to control-flow, as described by the specific questions that are presented below:

RQ11: Which spectrum is more effective to locate faults: data- or control-flow?

RQ12: What are the costs of execution associated with data- and control-flow?

RQ13: Which ranking metric is more effective for data- and control-flow spectra?

6.2 Experimental evaluation

In this section, we present our experimental design to compare the use of data- and control-flow spectra in fault localization. In what follows, we describe the programs and faults used in the experiment, the data collection procedure, the ranking metrics, and the analysis criteria used to answer our research questions.

6.2.1 Subject programs

We used five real-world Java programs for the experiment. JFreeChart, Commons Lang, Commons Math, and Joda-Time were obtained from the Defects4J database [Just et al., 2014]. The jsoup program was obtained from its source code repository, from which we identified the faults.

1 github.com/rjust/defects4j
2 github.com/jhy/jsoup
6.2.2 Procedure

This section presents the procedures used to collect the data and to compare the effectiveness of data- and control-flow spectra.

Treatments

We have two treatments in this study: data-flow (DF) and control-flow (CF) spectra. For DF, we use dua spectrum. For CF, we use line spectrum. We compare the fault localization effectiveness and efficiency of DF and CF using ten ranking metrics described below.

Data collection

We used Jaguar (see Section 5.2.1) to create the lists of suspicious duas and lines for each fault. We applied this procedure for all ten ranking metrics used in this evaluation: DRT, Jaccard, Kulczynski2, McCon, Minus, Ochiai, OP, Tarantula, Wong3, and Zoltar (see Section 2.4.1). Jaguar also collects the time spent to generate the suspiciousness lists, which we used to calculate efficiency for both spectra.

We created a script to check the position of the faults in the suspiciousness lists and to collect the time spent to create each list. The script allowed us to obtain the effectiveness and efficiency results without the need for manual intervention.

Selection of faults

We only selected faults for which data-flow spectrum was complete. The ba-dua tool (see Section 5.2.1), is unable to collect dua spectrum in two cases: (1) local duas (i.e., duas in which the definition and use occur in the same block); and (2) non-handled exceptions that are thrown during an execution. The first case arises because the ba-dua tool does not mark duas when the definition and use of a variable occur in the same block. Thus, we discarded bugs in single-block methods. The second case happens because ba-dua marks the exercised duas whenever a method is exited. Thus, a non-handled exception ends in a non-predictable way that is not covered by the tool. Such cases occur due to the lightweight manner ba-dua handles coverage information [de Araujo and Chaim, 2014]. Hence, we removed faults with incomplete data-flow spectrum for a fair comparison between dua and line spectra.

Table 6.1: Characteristics of the subject programs

<table>
<thead>
<tr>
<th>Program</th>
<th>KLOC</th>
<th>Faults</th>
<th>Test Cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>JFreeChart</td>
<td>96</td>
<td>26</td>
<td>2205</td>
</tr>
<tr>
<td>jsoup</td>
<td>10</td>
<td>38</td>
<td>468</td>
</tr>
<tr>
<td>Commons Lang</td>
<td>22</td>
<td>30</td>
<td>2245</td>
</tr>
<tr>
<td>Commons Math</td>
<td>85</td>
<td>43</td>
<td>3602</td>
</tr>
<tr>
<td>Joda-Time</td>
<td>28</td>
<td>26</td>
<td>4130</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td>163</td>
<td></td>
</tr>
</tbody>
</table>

used in this experiment. All programs are composed of real faults, and each version has a single fault.

JFreeChart is a chart library. Commons Lang a is library for the manipulation of standard Java classes. Joda-Time is date and time library. Commons Math is a library with mathematical and statistical functions. The Commons Math’s faults used in this study are different from those used in Chapter 4. Jsoup is an HTML parser.

Table 6.1 lists our subject programs, their number of lines of code (KLOC), the number of faulty versions, and the number of test cases.
Identification of faulty lines

We identified the faulty lines of each program version by comparing a fixing commit and its predecessor commit. We discarded changes in lines that contain non-executable code (e.g., comments), indentation changes, and empty lines.

There are faulty versions in which a fault is spread across several lines (i.e., multiple-line faults). That is, changes in more than one line were required to fix the bug. In this case, these lines were marked as the same bug.

For faults that occurred due to missing code, we marked the previous line of code as the bug site. However, if the previous line contains non-executable code (e.g., comments, blank lines, closing brackets), the line after the change was marked as the faulty one.

Measurement criteria

Effort budget. To answer our research questions, we measured the effectiveness of data- and control-flow to locate the faults by the absolute number of lines inspected until reaching the faults, i.e., effort budget, as we do in our other study (see Section 4.3.3). We restricted the number of lines inspected to 99 lines of code. Indeed, this is a huge limit since developers in practice inspect only the first picks of the suspiciousness lists [Kochhar et al., 2016; Parnin and Orso, 2011]. We measured the effectiveness of DF and CF for the following effort budgets: 5, 10, 20, 30, 40, 50, 60, 70, 80, 90, and 99.

For multiple-line faults, we considered that the bug is found when any of its faulty lines is reached. The rationale is that, if a developer reaches at least one of these lines, s/he locates the bug.

When ties occur, the worst case is considered. It means that if two or more lines have the same suspiciousness score, the position of the faulty line includes all tied lines since they have equal chances of being inspected.

The effort a developer allocates to an SFL technique may vary. Some check the first five lines and then give up when the faulty line is not one of them. Other developers are more persistent and go through the first 30 or 50 lines until s/he abandons the technique. The severity of a fault may also impact on the developers’ willingness to search for a bug. If the position of a fault in a ranking list is greater our effort budget (99 lines), we deem the fault as not found.

Time spent. We measured the efficiency of data- and control-flow spectra by the time spent to create the suspiciousness lists of each faulty version. The total overhead cost comprehends running each unit test, storing the spectrum data, calculating the suspiciousness of each element, and generating the suspiciousness list. For each program, we present the average time in seconds to execute Jaguar dua and line spectra in all its faults.

6.3 Results and discussion

This section presents our results and discussion regarding the effectiveness of data- and control-flow spectra, the efficiency of the two spectra, and the comparison of the ranking metrics for each spectrum. The results and discussion are guided by the research questions introduced in Section 6.1.

6.3.1 RQ11: Which spectrum is more effective to locate faults: data- or control-flow?

We compared the effectiveness of data- and control-flow spectra using the ten ranking metrics described in Table 2.2. The most suspicious duas and lines, ranked according to the metrics, are mapped onto lines to be inspected. Figures 6.1, 6.2, 6.3, 6.4 and 6.5 show the number of faults located by dua and line spectra for the different effort budgets and metrics.
Dua and line spectra presented similar behaviors for almost all ranking metrics, except for Wong3. They locate the same number of faults up to 7 investigated lines. After that, and up to 40 lines, dua spectrum locates more faults. The largest gap is located at 15 lines—for almost all metrics, dua locates 60 faults while line spectrum locates 40 faults. There is no difference between dua and line spectra for more than 40 lines inspected up to 80 lines. For more than 80 lines, line spectrum performs better. For Wong3, dua spectrum performs better after 5 lines, with a constant gap of 20 more faults located. This difference maintains almost constant until the end. However, Wong3 performs poorly, locating at most 60 faults when using dua and 40 with line spectrum. The other metrics locate more than 100 faults inspecting up to 99 lines using line spectrum. Dua spectrum locates about 80 faults by inspecting up to 20 lines using Ochiai, Jaccard, Kulczynski2, McCon, Op, and Minus. For the same metrics and budget, line spectrum locates about 60 faults.

We carried out a statistical analysis to compare the results obtained by data- and control-flow spectra with different ranking metrics. First, we applied the Anderson-Darling normality test [Anderson and Darling, 1954] to our data, which follows a non-normal distribution. Then, we used the Wilcoxon signed-rank test [Wilcoxon, 1945], which is suitable for the non-parametric distribution of our data. The confidence interval applied was 95%. Also, we used Cliff’s delta [Cliff, 1993] to
RESULTS AND DISCUSSION

Table 6.2 presents the results of the statistical analysis. The null hypothesis is that there is no difference in the effectiveness of data- and control-flow spectra to locate faults. The alternative hypothesis is that data-flow spectrum is more effective than control-flow spectrum—i.e., data-flow spectrum requires the examination of less suspicious lines to locate faults. We compared the number of blocks inspected (i.e., the positions where the faults were ranked) by dua and line spectra for each metric until the limit of 99 lines. If a fault is ranked above this limit, it is considered as not found.

The \( P \)-value column presents the results of the significance test. For all metrics, dua spectrum locates more faults inspecting less code with statistical significance (\( p \)-value \( \leq 5\% \)). The Effect Size column presents the effect size measured for all metrics, in which \( I \) means that the size is insignificant and \( S \) is a small effect (the effect size scale for Cliff’s delta is presented in Section 5.3.3). Except for Wong3, for which the effect size is small, the other metrics effect size is insignificant.

Table 6.2 also presents the median and the first quartile (\( Q1 \)) for both data-flow (DF) and control-flow (CF). Dua spectrum with Ochiai locates half of the faults by inspecting 25 lines while line spectrum needs 39 lines to reach the same amount of faults. The difference between medians
Figure 6.5: Effectiveness of dua and line spectra for $O^p$ and Minus

Table 6.2: Statistical results

<table>
<thead>
<tr>
<th>DF x CF</th>
<th>P-value (%)</th>
<th>Effect Size</th>
<th>Median DF</th>
<th>Median CF</th>
<th>Q1 DF</th>
<th>Q1 CF</th>
<th>B-T-W</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ochiai</td>
<td>1.59</td>
<td>0.032 (I)</td>
<td>25</td>
<td>39</td>
<td>9</td>
<td>15</td>
<td>73-50-40</td>
</tr>
<tr>
<td>Tarantula</td>
<td>0.59</td>
<td>0.035 (I)</td>
<td>27</td>
<td>39</td>
<td>10</td>
<td>15</td>
<td>75-48-40</td>
</tr>
<tr>
<td>Kulczynski2</td>
<td>1.66</td>
<td>0.023 (I)</td>
<td>26</td>
<td>36</td>
<td>9</td>
<td>11.5</td>
<td>74-49-40</td>
</tr>
<tr>
<td>McCon</td>
<td>1.66</td>
<td>0.023 (I)</td>
<td>26</td>
<td>36</td>
<td>9</td>
<td>11.5</td>
<td>74-49-40</td>
</tr>
<tr>
<td>Wong3</td>
<td>0.01</td>
<td>0.331 (S)</td>
<td>100</td>
<td>100</td>
<td>12</td>
<td>70</td>
<td>46-100-17</td>
</tr>
<tr>
<td>DRT</td>
<td>2.32</td>
<td>0.025 (I)</td>
<td>30</td>
<td>39</td>
<td>9</td>
<td>13.5</td>
<td>68-57-38</td>
</tr>
<tr>
<td>Minus</td>
<td>2.03</td>
<td>0.033 (I)</td>
<td>30</td>
<td>39</td>
<td>9</td>
<td>14.5</td>
<td>68-57-38</td>
</tr>
<tr>
<td>$O^p$</td>
<td>1.39</td>
<td>0.027 (I)</td>
<td>30</td>
<td>40</td>
<td>9</td>
<td>13.5</td>
<td>69-57-37</td>
</tr>
<tr>
<td>Jaccard</td>
<td>1.02</td>
<td>0.029 (I)</td>
<td>27</td>
<td>39</td>
<td>9</td>
<td>14.5</td>
<td>74-49-40</td>
</tr>
<tr>
<td>Zoltar</td>
<td>1.12</td>
<td>0.024 (I)</td>
<td>27</td>
<td>37</td>
<td>9</td>
<td>12.5</td>
<td>75-49-39</td>
</tr>
</tbody>
</table>

The results support the alternative hypothesis that data-flow improves fault localization by providing more detailed information. Dua spectrum leads the developer to inspect less code than line spectrum with statistical significance for all ranking metrics. However, the effect size is insignificant in almost all cases. Dua spectrum outperforms line spectrum in the range of 7 to 40 investigated lines. After that, there is no difference, or line spectrum performs better, which explains the insignificant effect size.

Indeed, if the faulty dua is not the first or second pick, the bug will hardly be located at the top 5 suspicious lines since duas are composed of a definition, a use, and possibly a source (in the case of p-use duas). Thus, each dua entails 2 or 3 lines to investigate. Overcoming line spectrum at the top 10 lines shows that dua spectrum singles out faulty code precisely. Additionally, dua spectrum demanded less code to be inspected for a large number of faults: 75 out of 163 using
Ochiai and Zoltar while line spectrum required the inspection of less code only for 40 and 39 faults, respectively, for the same ranking metrics—a difference of 87.5%.

Around 20 out of 163 (12%) faults can be located by investigating only the top 5 lines, using either dua or line spectra. On the other hand, one can reach 40 faults (25%) by investigating only 9 lines using dua spectrum. Furthermore, if the developer investigates up to 25 suspicious lines, s/he will reach half of the faults using dua spectrum. Using line spectrum, s/he will need to inspect 14 more lines to achieve the same result.

Some studies have shown that practitioners tend to give up on using an SFL technique if the fault is not located at the top 5 picks [Kochhar et al., 2016; Parnin and Orso, 2011]. However, in our user study (see Section 5.3.9), this fact does not occur. Since there is no difference between dua and line spectra up to 7 suspicious lines, the difference is too little to be useful in practice. Notwithstanding, sticking to only the first 5 lines is not practical either since too few faults can be located in this short range, in our study, only 12%. Data-flow spectrum can leverage SFL techniques. However, practitioners should be educated to inspect more code (e.g., from 10 to 25 lines to reach a quarter to half of the faults) to increase their chances to find bugs using SFL with dua spectrum.

Finally, dua spectrum brings different information for code inspection, such as suspicious variables, which may help to locate bugs. In our user study (see Section 5.3.8), some participants claimed that this kind of information should be provided. Future user studies can assess the usefulness of data-flow for SFL in practice.

6.3.2 RQ12: What are the costs of execution associated with data- and control-flow?

To understand the costs of using data- and control-flow spectra, we measured the time spent to use both spectra for SFL. Table 6.3 shows the results for the efficiency comparison. The JUnit, DF, and CF columns show the time spent (in seconds) for executing, respectively, only the JUnit tests, Jaguar with data-flow spectrum, and Jaguar with control-flow spectrum. These values consist of the average execution time for all versions of a project. We ran the experiments in a PC with an Intel Core i7 processor, 6GB of RAM running on Windows 7. At each execution, Jaguar runs the tests, collects the spectrum information, calculates the suspiciousness of all executed program entities for the ten ranking metrics, and creates a report file for each metric.

<table>
<thead>
<tr>
<th>Program</th>
<th>JUnit (s)</th>
<th>DF (s)</th>
<th>CF (s)</th>
<th>DF Over (%)</th>
<th>CF Over (%)</th>
<th>DF/CF (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>JFreeChart</td>
<td>22</td>
<td>88</td>
<td>36</td>
<td>309.9</td>
<td>70.1</td>
<td>143.7</td>
</tr>
<tr>
<td>jsoup</td>
<td>4</td>
<td>22</td>
<td>11</td>
<td>490.8</td>
<td>197.2</td>
<td>101.8</td>
</tr>
<tr>
<td>Commons Lang</td>
<td>18</td>
<td>89</td>
<td>33</td>
<td>538.7</td>
<td>131.8</td>
<td>164.3</td>
</tr>
<tr>
<td>Commons Math</td>
<td>144</td>
<td>515</td>
<td>265</td>
<td>301.8</td>
<td>89.2</td>
<td>94.5</td>
</tr>
<tr>
<td>Joda-Time</td>
<td>4</td>
<td>165</td>
<td>47</td>
<td>3298.0</td>
<td>881.9</td>
<td>245.0</td>
</tr>
</tbody>
</table>

The DF Over column shows the overhead to collect data-flow spectrum compared to the execution of JUnit. The CF Over column presents the overhead to collect control-flow spectrum compared to JUnit. The DF/CF column shows the ratio between the data-flow time and control-flow time. It indicates how much extra time is needed to execute Jaguar with data-flow compared to control-flow. The data-flow suspiciousness lists require from 94.5% to 245.09% more time to be generated than the control-flow lists. For example, generating a list of suspicious duas for the JFreeChart program requires 143.7% more time than generating a list of suspicious statements. However, in this case, the total time spent is 88 seconds.

The percentage of run-time overhead to execute data-flow obtained in our study differs a lot from the 38% overhead presented in the work of de Araujo and Chaim [2014], which describes the run-time costs to execute ba-dua on real programs. Such a discrepancy occurs due to the extra
work needed to run an SFL technique. For each test case, Jaguar stores information of the executed program entities and updates the coverage matrix. After the tests, it calculates the suspiciousness of each entity and, then, creates the suspiciousness lists.

Data-flow implies more information than control-flow: if on the one hand, it provides detailed data for fault localization; on the other hand, more time is needed to compute such data. Thus, control-flow is more efficient, providing fault localization information with less overhead.

However, the amount of time needed to create the lists of suspicious duas for the medium- and large-sized real programs used in this assessment is not prohibitive. For four out of five subjects, the time varies from 22 seconds to 2:45 minutes. Only for Commons Math, the time jumps to 8:35 minutes. For this particular program, line spectrum is also expensive to obtain: 4:25 minutes; even executing only JUnit takes 2:24 minutes in this case. Since dua spectrum can improve the fault localization effectiveness, the extra time to create the suspiciousness lists seems affordable for use in industrial settings.

6.3.3 RQ13: Which ranking metric is more effective for data- and control-flow spectra?

Figures 6.6 and 6.7 compare the effectiveness of all ranking metrics for dua and line spectra, respectively. For both spectra, the metrics perform very similarly, except for the Wong3 metric. In the smaller budgets, the difference among the ranking metrics is almost imperceptible.

For dua spectrum, Ochiai stands out slightly for budgets above 15. Below 15 lines, Jaccard, Kulczynski2, and McCon performed better. For line spectrum, Jaccard, Kulczynski2, and McCon are slightly better when inspecting up to 50 lines. Above 50 lines, Ochiai overcomes the other metrics. Also, Jaccard, Kulczynski2, and McCon have the same behavior throughout all budgets for both dua and line spectra.

We compared the ranking metrics that performed better for dua and line spectra. Overall, Jaccard, Kulczynski2, and McCon presented better performance. We checked the metrics that reached more bugs throughout all the budget range: Jaccard, Kulczynski2, and McCon performed better for small budgets for both spectra. For large budgets, the three metrics perform very close to Ochiai. Thus, figures 6.2a, 6.3b and 6.4a allow us to compare directly the performance of dua and line spectra for the better ranking metrics. This comparison indicates that dua spectrum overcomes line spectrum in most cases, especially in small budgets.

Figures 6.8 and 6.9 show the effectiveness of all ranking metrics for dua and line spectra, respectively. Each color represents the number of faults located in a budget. For example, in Figure 6.8,
26 faults were ranked between the 11th and 20th positions of their respective lists. These results indicate that the ranking metric does not have a great impact, especially focusing on the inspection of small budgets. The only exception is Wong3, which presented a poor performance for both spectra. Moreover, the best ranking metrics performed similarly for both dua and line spectra, which indicates that they can be used independently of spectra.

6.4 Threats to validity

The threats to the internal validity of this study are related to the Jaguar tool. To mitigate them, we manually checked the Jaguar’s outputs using small programs. Regarding the efficiency results, it is possible to optimize the Jaguar tool to reduce its run-time overhead for both spectra. Our chosen strategy aimed to get a first picture of the performance benefits from the recent promising data-flow spectrum tool (ba-dua). As we explain in Section 6.2.2, we selected bugs for which data-flow spectrum was complete. Thus, we excluded faults that throw exceptions without handling them and faults in single-block methods. Improvements in ba-dua can allow us to use such faults in future experiments.

All faults were found in their source code repositories. We assumed that each code change between a faulty version and its consecutive fixed version was made to fix the bug. We disregarded changes that do not affect the program behavior such as adding or removing empty lines and line indentation changes. For multiple-line faults, we considered the best case scenario. A fault is deemed as found when the top-ranked one of its lines is reached. However, this scenario occurs for both spectra. For tie cases, we considered the worst case, in which all lines with the same suspiciousness score of the faulty one are deemed to calculate its position.

Regarding threats to the external validity, we used five programs from different domains and sizes to expose the techniques to different scenarios. Although the programs utilized in our experimental assessment are quite heterogeneous, we caution the reader that the techniques may present different results for a different set of programs.

Regarding construct validity threats, we assessed the techniques using statistical tests and descriptive data. Our experiment was built to evaluate how quickly a technique will reach the fault site. One issue, though, deems the assumption that the ranked elements will be followed as they were presented, which may not happen in practice. Another issue is that reaching the bug site does not necessarily mean locating the defect, which is known as perfect bug detection [Parnin and Orso, 2011].
6.5 Final remarks

In this study, we carried out an experiment comparing the use of data-flow spectrum (definition-use associations—duas) and control-flow spectrum (lines) in fault localization. The results indicate that data-flow is more effective than control-flow, ranking faults in better positions in the SFL lists. Also, we aimed to verify whether new approaches to collect duas would allow the application of data-flow-based SFL in programs similar to those developed in the industry. We used 163 real faulty versions of five industry-like programs whose sizes vary from 10 to 96 KLOC.

Dua spectrum located more bugs inspecting less code for ten ranking metrics with statistical significance. Dua spectrum outperforms line spectrum in the range from 7 to 40 inspected lines. Recent studies have shown that practitioners expect the fault to be located in the first 5 most suspicious lines. Nevertheless, such a tight target is successful for only 12% of the faults in our experiment. SFL will hardly be adopted in practice with this small success rate. To take advantage of data-flow spectrum and SFL techniques for debugging in industrial settings, practitioners should be educated to inspect a few more code excerpts. Our results suggest that the inspection of up to 25 lines would significantly improve the number of faults hit by SFL techniques using data-flow spectrum.

Regarding efficiency, the overall time for the subjects of our experiment varies from 22 seconds to 8:35 minutes using data-flow spectrum, which suggests that its use is feasible for many programs developed by the industry. Thus, the hypothesis that data-flow generates more detailed information...
that improves fault localization was verified in our experiment. Although more time is needed to collect such information, the overhead is affordable for real-world software.

Data-flow is promising to improve both the contextualization and the practical use of SFL. To assess this issue, future user studies on SFL should consider the use of data-flow spectrum, especially information about suspicious variables, which may help practitioners to understand faulty behaviors.

In the next chapter, we present our conclusions regarding the studies presented in this thesis.

![Figure 6.9: Effectiveness of all ranking metrics for line spectrum](image)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ochia</td>
<td>2</td>
<td>6</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>jaccard</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>2</td>
<td>5</td>
<td>6</td>
<td>2</td>
<td>6</td>
<td>5</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>tarantula</td>
<td>2</td>
<td>5</td>
<td>8</td>
<td>4</td>
<td>4</td>
<td>6</td>
<td>4</td>
<td>6</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>zoltar</td>
<td>7</td>
<td>4</td>
<td>9</td>
<td>7</td>
<td>7</td>
<td>4</td>
<td>7</td>
<td>4</td>
<td>7</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>kulczynski2</td>
<td>9</td>
<td>7</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>14</td>
<td>12</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>dft</td>
<td>12</td>
<td>12</td>
<td>14</td>
<td>12</td>
<td>12</td>
<td>14</td>
<td>12</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>wong3</td>
<td>23</td>
<td>26</td>
<td>26</td>
<td>25</td>
<td>26</td>
<td>23</td>
<td>23</td>
<td>26</td>
<td>23</td>
<td>23</td>
<td>25</td>
</tr>
<tr>
<td>mccon</td>
<td>16</td>
<td>17</td>
<td>17</td>
<td>17</td>
<td>16</td>
<td>11</td>
<td>17</td>
<td>16</td>
<td>15</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>op</td>
<td>21</td>
<td>23</td>
<td>18</td>
<td>22</td>
<td>23</td>
<td>19</td>
<td>12</td>
<td>23</td>
<td>19</td>
<td>19</td>
<td>18</td>
</tr>
<tr>
<td>minus</td>
<td>23</td>
<td>23</td>
<td>18</td>
<td>23</td>
<td>23</td>
<td>12</td>
<td>23</td>
<td>19</td>
<td>19</td>
<td>19</td>
<td>18</td>
</tr>
</tbody>
</table>
Chapter 7

Conclusions

The studies carried out during this doctoral research were devised with focus on the practical adoption of SFL. Our contextualization approach (see Chapter 4) aims to provide more information for the debugging activity by bringing two levels of code information (methods and blocks) for fault localization. Thus, a developer can inspect a method and its internal blocks at the same time, which may help to understand bugs. Moreover, this study shows that the roadmaps reduce the amount of code to inspect.

Our user study (see Chapter 5) aimed to understand how SFL is used by practitioners, whether SFL improves debugging, and whether they consider using SFL in the future. Fault localization techniques will only be useful if their target users really intend to use them in practice. For this purpose, techniques should improve the debugging performance.

The study about the use of data-flow spectrum for SFL (see Chapter 6) assessed whether it improves the fault localization performance and what are the execution costs associated with data-flow for SFL. Data-flow provides additional information about suspicious variables, and also how these variables are defined and used in the code. Such information may also help developers to understand bugs.

In the following sections, we present our main findings and how they relate to each other. We conclude this chapter by presenting directions for future work.

7.1 Findings and implications

Proposing new strategies to improve SFL

Ranking metrics have a pivotal role in SFL since they are responsible for calculating the suspiciousness of program elements. However, there is a limit to the effectiveness that ranking metrics can provide for fault localization; our study of Chapter 6 shows that metrics that reached more faults had an identical performance. Previous studies also show the similar performance of different metrics [Debroy and Wong, 2011; Le et al., 2013a; Pearson et al., 2017; Xie et al., 2013].

As we show in Chapter 4, the combination of roadmaps with information of methods and blocks, and filtering strategies improved the fault localization performance of existing ranking metrics by reducing the amount of code to be inspected.

Some of the most recent SFL studies have used method-level information for fault localization [Kochhar et al., 2016; Le et al., 2016; Mariani et al., 2011]. Indeed, methods are the lowest code level that contains a complete logic of programs’ functionalities. Methods may be more understandable than isolated statements. Differently from these studies, we combine information of methods and their most suspicious code blocks, which may help developers to focus on possible fault causes inside a method.

Filtering strategies, like those proposed in our study (see Section 4.2.3), seem to be a way to help SFL techniques to narrow down the amount of code for inspection during the fault localization process. Those strategies may also be useful when using data-flow spectrum. Future studies can
investigate this issue.

Assessing the practical use of SFL

The results of our user study (see Chapter 5) show that SFL improves the developers’ performance on finding bugs. More developers located the bugs using the Jaguar tool than without using it. However, the time spent to find the bugs was not reduced using SFL. These results indicate that SFL is useful in practice.

Regarding the developers’ behavior using SFL, our user study shows that developers do not inspect large amounts of suspicious code. They focused more on the first picks, inspecting at most 20 different program entities. This fact shows the importance of proposing new ways to reduce the amount of code to be inspected to locate bugs. By doing this, we can develop techniques more suitable for practical use.

Also, our study corroborates findings of previous studies [Parnin and Orso, 2011; Xia et al., 2016]. Developers do not identify a bug immediately after click on the faulty entity. In most cases, they inspect the faulty entity more than once to identify the fault. Also, they do not follow the SFL lists in their exact order, but the first picks are often investigated.

Additionally, our user study shows that developers that inspected the faulty code had a higher chance of finding bugs. Although it seems obvious, we show that this relationship occurs with statistical significance. Then, we show that SFL led more developers to inspect the faulty code, which resulted in more bugs found by developers using our SFL tool.

The participants of our user study deemed SFL as useful, showing the intention to use SFL in the future. Even those that did not find bugs using SFL showed this intention. Thus, it is important to disclose automated debugging techniques aiming at their practical use. It is important to develop tools that are easy to use, and that can be used in real software projects. Integration with IDEs can also facilitate the adoption of those tools.

Assessing the use of data-flow for SFL

As data-flow (dua spectrum) provides more detailed information about the relationships of variables through the code, its use for SFL can help to pinpoint more precisely the faulty code entities, improving the SFL results.

Indeed, the results of our study show that dua spectrum improves the fault localization effectiveness. Although a single dua is spread over two or three lines of code, dua spectrum ranked more faults in the highest positions than line spectrum.

Data-flow requires more time and space to process its information. However, the extra time is not prohibitive, even for the large programs we used in the evaluation. It is suitable to be used in commodity computers. The use of large programs was possible due to the development of a new lightweight approach to collect dua spectra [Chaim and de Araujo, 2013], which is implemented by the ba-dua tool. By obtaining data-flow spectrum at lower costs of execution, SFL techniques can devise new strategies to explore such information to improve their results.

Using absolute values to assess SFL

Most studies evaluate their techniques using the percentage of code inspected to reach the bugs. When coping with real programs, the percentage of code can mislead the fault localization results: a technique that finds most bugs inspecting 1% of the code may lead to the inspection of 500 lines in a medium-sized program with 50 KLOC.

In our studies of Chapters 4 and 6, we assessed the effectiveness of the techniques using the absolute number of blocks or lines that are inspected until reaching the bugs. Some recent studies have used this approach [Le et al., 2015a; Sohn and Yoo, 2017; Steimann and Bertschler, 2009]. Using absolute values to measure fault localization effectiveness can provide more realistic results about the techniques’ performance. Moreover, our user study (Chapter 5) shows that developers do not inspect more than 20 distinct suspicious elements.
7.2 Future work

There are several possibilities for further studies regarding fault localization and the approaches presented in this research.

In future work, we intend to perform user studies with the roadmaps and filtering strategies. We also intend to assess roadmaps and filtering strategies along with other spectra such as definition-use associations (duas) and branches. Dua spectrum seems particularly interesting because it indicates both suspicious lines and variables for investigation.

There are other ways of using contextual information to improve fault localization: code metrics and change history may be used for this purpose. New filtering strategies can also be explored, taking into account other characteristics of programs.

We also intend to perform experiments in which developers have previous knowledge of the code to see how useful SFL is in this scenario. Exploratory studies and case studies should also be conducted, assessing the use of SFL in software companies.

The evaluation of other real programs is also necessary. Most of the real programs currently used to evaluate SFL techniques are frameworks and libraries. End-user programs should also be assessed to comprehend the performance of SFL in those programs better. We also intend to conduct experiments to evaluate how data-flow spectrum behaves in programs with multiple-faults.

Another future issue regarding the use of contextual information for fault localization is the programming language. It is necessary to investigate how fault localization techniques perform in programs written in other languages, which have different structures and belong to different programming paradigms.
Finally, there is a need for understanding the influence of program characteristics (e.g., test suite size, fault types, cohesion, coupling) on the performance of fault localization techniques, which should be explored in future research.
Appendix A

User study: Questionnaires, training, instructions, and consent letter

A.1 Pre-task questionnaire

This questionnaire was designed as online form to be applied before the experiments. Our intention is to gather information about the development experience of the participants to balance the distribution through the experimental groups and to assess the impact of experience in the debugging performance. We wrote a Portuguese version of both questionnaires for this experiment.

How many years have you been programming?*
Help text: Consider your educational and professional experience in any programming language
Question type: Scale
0 to 10–10 years or more

How many years have you been programming for software companies?*
Question type: Scale
0 to 10–10 years or more

How many years have you been programming in Java?*
Question type: Scale
0 to 10–10 years or more

What is your current expertise level in Java?*
Question type: Multiple choice
- None
- Basic - sporadic use / used for developing a few small programs
- Intermediate - moderate use / used for developing a few large programs
- Advanced - frequent use / used on several large programs

How many years have you been programming using Eclipse, Netbeans, and/or IntelliJ IDEA?*
Question type: Scale
0 to 10–10 years or more

What is your current expertise level in Eclipse, Netbeans, and/or IntelliJ IDEA?*
Question type: Multiple choice

* means a required question.
• None
• Basic - sporadic use / used for developing a few small programs
• Intermediate - moderate use / used for developing a few large programs
• Advanced - frequent use / used on several large programs

How many years have you been testing using JUnit and/or TestNG?*
Question type: Scale
0 to 10–10 years or more

What is your current expertise level in JUnit and/or TestNG?*
Question type: Multiple choice
• None
• Basic - theoretical knowledge / studying or training use
• Intermediate - moderate use / used for developing a few projects
• Advanced - frequent use

A.2 Pos-task questionnaire

This questionnaire was designed as online form to be applied after the experiments. Our intention is to gather information about the Jaguar usability and acceptance, participants’ impressions on using the technique, and their performance in the debugging tasks.

What is your ID number?*\(^2\)
Help text: The number generated in the previous questionnaire
Question type: Text

What project did you debug using the Jaguar tool?*
Question type: Multiple choice
• jsoup
• XStream

Can you explain what caused the bug of the jsoup project?*
Help text: If you don’t know simply answer “I don’t know”
Question type: Paragraph text

What is the name of the class that contains the bug of the jsoup project?*
Help text: If you don’t know simply answer “Not found”
Question type: Text

What is the name of the method that contains the bug of the jsoup project?*
Help text: If you don’t know simply answer “Not found”
Question type: Text

What is the number of the line that contains the bug of the jsoup project?*
Help text: If you don’t know simply answer “Not found”

\(^2\) means a required question.
Question type: Text

Please describe how you would fix the bug of the jsoup project*
Help text: If you don’t know simply answer “I don’t know”
Question type: Paragraph text

What is the difficulty of finding the bug of the jsoup project*
Question type: Scale
Easy–1 to 7–Hard

Have you had any previous contact with the code of jsoup?*
Question type: Multiple choice
• Yes
• No

Can you explain what caused the bug of the XStream project?*
Help text: If you don’t know simply answer “I don’t know”
Question type: Paragraph text

What is the name of the class that contains the bug of the XStream project?*
Help text: If you don’t know simply answer “Not found”
Question type: Text

What is the name of the method that contains the bug of the XStream project?*
Help text: If you don’t know simply answer “Not found”
Question type: Text

What is the number of the line that contains the bug of the XStream project?*
Help text: If you don’t know simply answer “Not found”
Question type: Text

Please describe how you would fix the bug of the XStream project*
Help text: If you don’t know simply answer “I don’t know”
Question type: Paragraph text

What is the difficulty of finding the bug of the XStream project*
Question type: Scale
Easy–1 to 5–Hard

Have you had any previous contact with the code of XStream?*
Question type: Multiple choice
• Yes
• No

Did the Jaguar tool help you to find the bug?*
Help text: Consider only the bug in which you have used the tool
Question type: Multiple choice
• I found the bug without using the tool
• The tool was not very useful to find the bug
• The tool was somehow useful to find the bug
• The tool was very useful to find the bug
• I could not have found the bug without the tool

Describe how the tool helped you to find the bug*  
Help text: Otherwise, simply answer “It didn’t help me”  
Question type: Paragraph text

What information provided by the tool was more useful to perform the debugging task?*  
Help text: Otherwise, simply answer “None”  
Question type: Paragraph text

The organization provided by the tool was useful?*  
Help text: Information structured in methods or statements. Information sorted by suspiciousness scores.  
Question type: Scale  
Useless–1 to 7–Useful

What is your opinion about the color scheme provided by the tool?*  
Question type: Scale  
Useless–1 to 7–Useful

What is your opinion about the list provided by the tool?*  
Question type: Scale  
Useless–1 to 7–Useful

What is your opinion about the slider provided by the tool?*  
Question type: Scale  
Useless–1 to 7–Useful

What is your opinion about the text search filter provided by the tool?*  
Question type: Scale  
Useless–1 to 7–Useful

What other information would you like to see in this tool?  
Question type: Paragraph text

Do you have any other comments, observations, or suggestions?  
Question type: Paragraph text

Assuming I had access to Jaguar, I intend to use it*  
Question type: Scale  
Strongly disagree–1 to 7–Strongly agree

Given that I had access to Jaguar, I predict that I would use it*  
Question type: Scale  
Strongly disagree–1 to 7–Strongly agree

Using Jaguar improves my performance in the debugging task*  
Question type: Scale
Strongly disagree–1 to 7–Strongly agree

**Using Jaguar in the debugging task increases my productivity***
Question type: Scale
Strongly disagree–1 to 7–Strongly agree

**Using Jaguar enhances my effectiveness in the debugging task***
Question type: Scale
Strongly disagree–1 to 7–Strongly agree

**Overall, I find Jaguar useful to perform the debugging task***
Question type: Scale
Strongly disagree–1 to 7–Strongly agree

**Learning to use Jaguar is easy for me***
Question type: Scale
Strongly disagree–1 to 7–Strongly agree

**Interacting with Jaguar does not require a lot of mental effort***
Question type: Scale
Strongly disagree–1 to 7–Strongly agree

**Overall, I find Jaguar easy to use***
Question type: Scale
Strongly disagree–1 to 7–Strongly agree

**I find easy to get Jaguar to do what I want it to do***
Question type: Scale
Strongly disagree–1 to 7–Strongly agree

**Which of the following options have you been used in your debugging tasks?***
Question type: Checkboxes
- Breakpoints
- Print statements
- Testing results
- Stack traces
- Bug reports
- Other

**Do you have any experience using some fault localization technique?***
Help text: A fault localization technique is one that automatically indicate suspicious code excerpts which may contain a bug.
Question type: Multiple choice
- Yes
- No

**If you answered yes in the question above, what fault localization technique have you been used?***
Question type: Text
A.3 Training - About the Jaguar tool

This document was written for the training on using Jaguar. There are two versions of the document for the method list and the statement list. The version below is for the method list.

Jaguar is a fault localization tool for Eclipse. The tool shows methods and that are more suspicious of being buggy. Jaguar uses the JUnit results to pinpoint bug candidates.

Figure A.1: Eclipse’s Jaguar plugin

Figure A.1 shows the Jaguar plugin at the right side of Eclipse. Figure A.2 shows details of the tool - Area 1 has the most suspicious methods, which are placed in descending order of score. The score indicates how likely the method is to contain the bug. The higher the score, the more suspicious it is.

The colors represent four suspiciousness score levels: red represents the most suspicious bug candidates, orange those high scores, yellow are the moderate bug candidates, and green the least suspicious ones.

Navigation

When you click on a method in Jaguar, the file that contains this method is opened in the Editor area (see Figure A.1). Jaguar also provides a slider (shown in Area 3) to filter methods that are equal or higher than the selected score. The tool also provides a text search filter, shown in Area 2, which can be used to search for specific terms in the code. Both the slider and the text filter return all methods within the requested values.
Figure A.2: Jaguar view

**Video**

We prepared a video which shows an example of Jaguar in use. You can open the file `jaguar.mp4` located on the desktop area to see the video\(^3\).

**Try Jaguar**

On the desktop area, click on the Jaguar example icon to explore the tool before starting your assigned tasks. The jtopas program is included as an example of use.

1. Right-click on the jtopas project
2. Select the option Jaguar > Run Jaguar
3. The plugin will be opened on the right side of Eclipse
4. Click on the methods to see their code.
5. The bug is in the method `AbstractTokenizer.isKeyword(int, int)` at line 1559:

   ```java
   String keyword = new String(_inputBuffer, startingAtPos-_rangeStart, length-1);
   ```

   The value assigned to instantiate keyword should be `length` instead of `length - 1`.

---

\(^3\)The videos used in the experiments are available in the study’s repository at: github.com/saeg/user-study-sfl
6. Close Eclipse and read the file instructions.pdf to get started on your tasks.

A.4 Experiment instructions

There are four versions of this document, one for each experimental group. The differences in the documents are the order of the debugging tasks. The version above is for groups 1 and 5. For the other groups, the instructions are similar, changing the order of the tasks as shown in Table 5.1.

Start the experiment by Task 1. When you finish it, go to Task 2.

Task 1 - Debugging the jsoup project with Jaguar

jsoup is an HTML parser library.

There are test cases that fail due to a bug, which can be seen running the tests.

1. Click on the Eclipse icon.

2. To open Jaguar, right-click on the jsoup project at the Project Explorer area, select Jaguar > Run Jaguar.

3. The tool will appear on the right side of Eclipse. Please do not change the Eclipse perspective. Jaguar was configured for the Resource perspective only.

4. To start the study, click on the bug button at the top of the Jaguar tool. Type your ID generated in the questionnaire about technical knowledge. The information will be loaded on the tool's view.

5. You can use both the Jaguar tool and any other Eclipse resource for this task. To run the tests, right-click on the jsoup project at the Project Explorer area, select Run As > JUnit Test. In case you have to select the preferred JUnit launcher, you must choose the Eclipse JUnit Launcher instead of Jaguar JUnit Delegate Launcher.

6. You have 30 minutes to complete this task. Although the tool logs the time, it does not provide a timer, so we kindly ask you to manage the time of each task by yourself.

7. Please write down any changes you made in the code to fix the bug, clearly indicating class name, method name, and line number for these changes. You will need this information when answering the final questionnaire.

8. When you finish the task, either because you found the bug or because you ran out of time, click on the stop button at the top of the Jaguar tool. When you do so, the tool will close, and you will be able to start the following task. Please do not close Eclipse.

Task 2 - Debugging the XStream project without Jaguar

XStream is a library to serialize objects to XML and back again.

A bug causes errors in some test cases, which can be seen running the tests.

1. To start the task, click on the bug button at the top of the Project Explorer area. You can use any Eclipse resource to search for the bug. The Jaguar tool is not available for this task.
A.5 Consent letter

The consent letter was sent by email, inviting the participants to take part in the experiment and explaining the selection criteria for the participants and all procedures to perform the experiment.

Informed Consent Letter

Project title: How do developers search for bugs using fault localization tools?

Investigators: Higor Amario de Souza, Ph.D. candidate, Professor Marcos L. Chaim, and Professor Fabio Kon.

Dear Student/Professional,

You are being asked to be a volunteer in a research study that aims to investigate how developers debug Java programs using a fault localization tool in the Eclipse IDE. The tool is open source and will be available for use soon. Your participation in this experiment will be valuable to design a tool for use in real development settings.

This study will be conducted remotely through an already configured virtual machine, and can be performed anywhere and whenever you prefer. Instructions to download the virtual machine are at the end of this document.

This study is part of a doctoral research project and is being conducted by Professor Fabio Kon and Professor Marcos Lordello Chaim, project supervisors, and Higor A. de Souza, Ph.D student at University of São Paulo. This study is funded by the São Paulo Research Foundation (FAPESP), protocol 2014/23030-5, Brazil.

If you would like to take part in this study, and we hope that you will, please read this Informed Consent Letter. We kindly ask you to do the experiment until <date>.

Inclusion/Exclusion Criteria:
We are searching for participants who have experience in development, testing and debugging of
Java programs using the Eclipse IDE and JUnit. Experience with Netbeans IDE, IntelliJ IDEA or TestNG can also be considered due to their similarity with Eclipse and JUnit. We expect to enroll 32 subjects in this study.

Procedures:
If you agree to participate in this study, you will be asked to take part in two debugging tasks. First, you will fill out a questionnaire about your programming knowledge, which will provide you a link for downloading a virtual machine (VM). This VM is already configured and ready for use. It also contains the training material and the instructions to perform the experiment. Therefore, you can perform the tasks whenever and wherever it is convenient for you. After completing the tasks, you will have to complete another questionnaire, which contains multiple choices and open ended questions to get your opinion about the experiment, the tasks, and the tool.

During the experiment, Eclipse will track your actions while performing the tasks. This tracking data will be sent to us automatically at the end of the second task. The experiment should take approximately 2 hours and 10 minutes. This time includes 5 minutes to fill out the first questionnaire, 20 minutes to download the virtual machine, 20 minutes for training, 1 hour to perform the experiment (30 minutes for each debugging task), and 15 minutes to fill out the second questionnaire.

Risks or Discomforts:
The risks involving your participation in this study are no greater than those existing in your daily programming activities.

Benefits:
Your participation will be a worthy addition to our research. You will probably have no direct benefits from joining this study, although the tool evaluated in the experiment will be available for you to use as an open source program. We hope that our findings will lead to a better understanding of the debugging process and to the development of better tools to support debugging activities.

Compensations/Costs to You:
There is no compensation for participating in this study. There are no costs to you, other than your time, for being in this study.

Anonymity, Confidentiality and Privacy:
The following procedures will ensure your anonymity, privacy and confidentiality. In the questionnaires, you will not be asked any personal information. Thus, the questions about your experience with software development techniques will not be linked to you. When filling out the first questionnaire, a numeric ID will be randomly created for you. You must enter your ID in the experiment and also in the second questionnaire. This ID will be used to associate each tracking data to its respective questionnaires. The questionnaires and tracking data will be accessed only by the study staff and will be kept in password protected archives for archival purposes. To make sure that this research is being performed in a proper way, the Research Board of the Department of Computer Science from the Institute of Mathematics and Statistics (USP) may review study records.

In Case of Injury/Harm:
If you experience any problem, harm or discomfort as a result of being in this study, please contact Higor Amario de Souza by email ( <email> ). Neither Higor Amario de Souza nor the Institute of Mathematics and Statistics have made provision for payment of costs associated with any injury resulting from participation in this study.

Participant Rights:
• Your participation in this study is voluntary.
• You may refuse to participate or withdraw at any time.
• Non-participation will not affect your grades or academic standing.

Questions about the Study:
If you have any questions about the study, do not hesitate to contact Higor A. de Souza (<email>), Marcos L. Chaim (<email>) or Fabio Kon (<email>).

Consent:
If you are willing to participate in this study, please follow the link provided in the instructions for the study to answer the questionnaire about your experience. After completing this questionnaire, you will receive a link for downloading a virtual machine (VM).

By completing the online questionnaires and performing the tasks, you indicate your consent to be in the study and that you have read the information given in this consent form.

Ethical approval for this study has been obtained from the Research Board of the Department of Computer Science at the Institute of Mathematics and Statistics (USP).

The results from this study will be published in a scientific journal and/or conference paper. You can also contact us for more information about the results at any time.

Thank you in advance for your time and collaboration.

Sincerely,
Professor Fabio Kon
Professor Marcos Lordello Chaim
Higor Amario de Souza, Ph.D. Student

Instructions for the study
Please fill out the questionnaire about your experience in development at <link>. After completing the questionnaire, you will receive a link for the virtual machine, your ID number, your login and password to access the VM. This VM contains all the required settings, as well as the instructions to use the fault localization tool and to perform the tasks. The VM runs on VirtualBox, which is available at <link> for Linux, OS X, and Windows.

1. Extract the VM file.

2. Click on the lubuntu-method.vbox (or lubuntu-line.vbox) file to open the VM in VirtualBox. The VM is set to run using 1GB RAM, thus your machine must have at least 2GB RAM to execute the VM. In case you have less than 2GB RAM, it is also possible to change the VM memory settings to reduce the VM’s RAM size.

3. For a better visualization, we recommend a minimum resolution of 1152 x 864. The screen resolution is generally adjusted automatically by VirtualBox.

4. Type your username and password.

5. The initial training instructions are in jaguar-tool.pdf on the Desktop area.

6. After finishing the tasks, please fill out the final questionnaire, available at <link>.

If you have any difficulties to download or use the VM please contact us by email.
Bibliography


