HENRIQUE LEMOS RIBEIRO

On the use of control- and data-flow in fault
localization

São Paulo

2016

HENRIQUE LEMOS RIBEIRO

**On the use of control- and data-flow in fault
localization**

Área de concentração: Engenharia da
computação

Versão corrigida contendo as alterações solicitadas pela comissão julgadora em 19 de Agosto de 2016. A versão original encontra-se em acervo reservado na Biblioteca da EACH-USP e na Biblioteca Digital de Teses e Dissertações da USP (BDTD), de acordo com a Resolução CoPGr 6018, de 13 de outubro de 2011.

Supervisor: Prof. Dr. Marcos Lordello Chaim

São Paulo

2016

Autorizo a reprodução e divulgação total ou parcial deste trabalho, por qualquer meio convencional ou eletrônico, para fins de estudo e pesquisa, desde que citada a fonte.

CATALOGAÇÃO-NA-PUBLICAÇÃO

(Universidade de São Paulo. Escola de Artes, Ciências e Humanidades. Biblioteca)

Ribeiro, Henrique Lemos

On the use of control- and data-flow in fault localization /
Henrique Lemos Ribeiro ; orientador, Marcos Lordello Chaim. –
São Paulo, 2016.

94 p. : il.

Dissertação (Mestrado em Ciências) - Programa de Pós-
Graduação em Sistemas de Informação, Escola de Artes,
Ciências e Humanidades, Universidade de São Paulo
Versão corrigida

1. Engenharia de software. I. Chaim, Marcos Lordello,
orient. II. Título

CDD 22.ed.– 005.1

Dissertação de autoria de Henrique Lemos Ribeiro, sob o título “**On the use of control-and data-flow in fault localization**”, apresentada à Escola de Artes, Ciências e Humanidades da Universidade de São Paulo, para obtenção do título de Mestre em Ciências pelo Programa de Pós-graduação em Sistemas de Informação, na área de concentração Metodologia e Técnicas da Computação, aprovada em _____ de _____ de _____ pela comissão julgadora constituída pelos doutores:

Prof. Dr. _____

Presidente

Instituição: _____

Prof. Dr. _____

Instituição: _____

Prof. Dr. _____

Instituição: _____

Prof. Dr. _____

Instituição: _____

*Dedico aos meus pais, Toninho e Lucia e minha irmã Gabriela que sempre me apoiaram
de várias maneiras nessa importante etapa da minha vida.*

Acknowledgements

Agradeço a todos que fizeram e fazem parte do grupo SAEG, que me ajudaram direta e indiretamente no desenvolvimento deste trabalho. Também aos meus amigos e parentes que me ajudaram não exatamente na parte acadêmica, mas com certeza em outras áreas que me influenciaram positivamente para a conclusão deste projeto.

“Yes and no...this or that...one or zero. On the basis of the elementary two-term discrimination, all human knowledge is built up. The demonstration of this is the computer memory which stores all its knowledge in the form of binary information. It contains ones and zeros, that’s all.

Because we are unaccustomed to it, we don’t usually see that there’s a third possible logical term equal to yes and no which is capable of expanding our understanding in an unrecognized direction. We don’t even have a term for it, so I will have to use the Japanese mu.

Mu means ‘no thing’. Like ‘Quality’ it points outside the process of dualistic discrimination. Mu simply says, ‘No class; not one, not zero, not yes, not no’. It states that the context of the question is such that a yes or no answer is in error and should not be given. ‘Unask the question’ is what it says.

Mu becomes appropriate when the context of the question becomes too small for the truth of the answer. When the Zen monk Joshu was asked whether a dog had a Buddha nature he said ‘Mu’, meaning that if he answered either way he was answering incorrectly. The Buddha nature cannot be captured by yes-or-no questions.”

(Zen and the Art of Motorcycle Maintenance by Robert M. Pirsig)

Abstract

RIBEIRO, Henrique Lemos. **On the use of control- and data-flow in fault localization**. 2016. 94 p. Dissertation (Master of Science) – School of Arts, Sciences and Humanities, University of São Paulo, São Paulo, 2016.

Testing and debugging are key tasks during the development cycle. However, they are among the most expensive activities during the development process. To improve the productivity of developers during the debugging process various fault localization techniques have been proposed, being Spectrum-based Fault Localization (SFL), or Coverage-based Fault Localization (CBFL), one of the most promising. SFL techniques pinpoints program elements (e.g., statements, branches, and definition-use associations), sorting them by their suspiciousness. Heuristics are used to rank the most suspicious program elements which are then mapped into lines to be inspected by developers. Although data-flow spectra (definition-use associations) has been shown to perform better than control-flow spectra (statements and branches) to locate the bug site, the high overhead to collect data-flow spectra has prevented their use on industry-level code. A data-flow coverage tool was recently implemented presenting on average 38% run-time overhead for large programs. Such a fairly modest overhead motivates the study of SFL techniques using data-flow information in programs similar to those developed in the industry. To achieve such a goal, we implemented Jaguar (*Java coveraGe faUlt locAlization Ranking*), a tool that employ control-flow and data-flow coverage on SFL techniques. The effectiveness and efficiency of both coverages are compared using 173 faulty versions with sizes varying from 10 to 96 KLOC. Ten known SFL heuristics to rank the most suspicious lines are utilized. The results show that the behavior of the heuristics are similar both to control- and data-flow coverage: Kulczynski2 and Mccon perform better for small number of lines investigated (from 5 to 30 lines) while Ochiai performs better when more lines are inspected (30 to 100 lines). The comparison between control- and data-flow coverages shows that data-flow locates more defects in the range of 10 to 50 inspected lines, being up to 22% more effective. Moreover, in the range of 20 and 100 lines, data-flow ranks the bug better than control-flow with statistical significance. However, data-flow is still more expensive than control-flow: it takes from 23% to 245% longer to obtain the most suspicious lines; on average data-flow is 129% more costly. Therefore, our results suggest that data-flow is more effective in locating faults because it tracks more relationships during the program execution. On the other hand, SFL techniques supported by data-flow coverage needs to be improved for practical use at industrial settings.

Keywords: software engineering, fault localization, data-flow, control-flow

Resumo

RIBEIRO, Henrique Lemos. **Sobre o uso de fluxo de controle e de dados para a localizao de defeitos**. 2016. 94 f. Dissertação (Mestrado em Ciências) – Escola de Artes, Ciências e Humanidades, Universidade de São Paulo, São Paulo, 2016.

Teste e depuração são tarefas importantes durante o ciclo de desenvolvimento de programas, no entanto, estão entre as atividades mais caras do processo de desenvolvimento. Diversas técnicas de localização de defeitos têm sido propostas a fim de melhorar a produtividade dos desenvolvedores durante o processo de depuração, sendo a localização de defeitos baseados em cobertura de código (*Spectrum-based Fault Localization (SFL)*) uma das mais promissoras. A técnica SFL aponta os elementos de programas (e.g., comandos, ramos e associações definição-uso), ordenando-os por valor de suspeição. Heurísticas são usadas para ordenar os elementos mais suspeitos de um programa, que então são mapeados em linhas de código a serem inspecionadas pelos desenvolvedores. Embora informações de fluxo de dados (associações definição-uso) tenham mostrado desempenho melhor do que informações de fluxo de controle (comandos e ramos) para localizar defeitos, o alto custo para coletar cobertura de fluxo de dados tem impedido a sua utilização na prática. Uma ferramenta de cobertura de fluxo de dados foi recentemente implementada apresentando, em média, 38% de sobrecarga em tempo de execução em programas similares aos desenvolvidos na indústria. Tal sobrecarga, bastante modesta, motiva o estudo de SFL usando informações de fluxo de dados. Para atingir esse objetivo, Jaguar (*JJava coveraGe faUlt locAlization Ranking*), uma ferramenta que usa técnicas SFL com cobertura de fluxo de controle e de dados, foi implementada. A eficiência e eficácia de ambos os tipos de coberturas foram comparados usando 173 versões com defeitos de programas com tamanhos variando de 10 a 96 KLOC. Foram usadas dez heurísticas conhecidas para ordenar as linhas mais suspeitas. Os resultados mostram que o comportamento das heurísticas são similares para fluxo de controle e de dados: Kulczynski2 e Mccon obtêm melhores resultados para números menores de linhas investigadas (de 5 a 30), enquanto Ochiai é melhor quando mais linhas são inspecionadas (de 30 a 100). A comparação entre os dois tipos de cobertura mostra que fluxo de dados localiza mais defeitos em uma variação de 10 a 50 linhas inspecionadas, sendo até 22% mais eficaz. Além disso, na faixa entre 20 e 100 linhas, fluxo de dados classifica com significância estatística melhor os defeitos. No entanto, fluxo de dados é mais caro do que fluxo de controle: leva de 23% a 245% mais tempo para obter os resultados; fluxo de dados é em média 129% mais custoso. Portanto, os resultados indicam que fluxo de dados é mais eficaz para localizar os defeitos pois rastreia mais relacionamentos durante a execução do programa. Por outro lado, técnicas SFL apoiadas por cobertura de fluxo de dados precisam ser mais eficientes para utilização prática na indústria.

Palavras-chaves: engenharia de software, localização de defeitos, fluxo de dados, fluxo de controle

List of Figures

Figure 1 – Code of <code>max</code> program	23
Figure 2 – Control-flow graph of <i>max</i> program	25
Figure 3 – Control-flow graph of the max program including data-flow information	27
Figure 4 – Slices of variable <code>max</code> at line 11 when running <code>max([4,3,2],3)</code>	29
Figure 5 – Coverage of <code>max</code> function with Tarantula heuristic	32
Figure 6 – Inclusion and exclusion criteria result	37
Figure 7 – Inclusion and exclusion criteria result by database	38
Figure 8 – Distribution of the type of data-flow techniques over all papers	42
Figure 9 – Programming languages used by each approach over the years	43
Figure 10 – Jaguar architecture	52
Figure 11 – Jaguar View - Flat	56
Figure 12 – Jaguar View - Hierarchical	57
Figure 13 – Effectiveness of heuristics using various budgets for control-flow. . . .	65
Figure 14 – Effectiveness of heuristics using various budgets for data-flow. . . .	66

List of Tables

Table 1 – All nodes and all edges of <i>max</i> program.	26
Table 2 – All definition-use associations of the <i>max</i> program.	28
Table 3 – SFL Coefficients	31
Table 4 – Heuristics for fault localization	32
Table 5 – Test Suite	33
Table 6 – Data base research result	37
Table 7 – Related Work Summary - I	38
Table 8 – Programs characteristics	60
Table 9 – Program versions	61
Table 10 – Control-flow versus data-flow effectiveness	67
Table 11 – Heuristic versus heuristic: results for control-flow	68
Table 12 – Heuristic versus heuristic: results for data-flow	68
Table 13 – Control-flow and Data-flow efficiency for each project	69
Table 14 – Control-flow and Data-flow located faults	72
Table 15 – Heuristic versus heuristic — Control-flow — Budget 5	88
Table 16 – Heuristic versus heuristic — Control-flow — Budget 10	89
Table 17 – Heuristic versus heuristic — Control-flow — Budget 20	89
Table 18 – Heuristic versus heuristic — Control-flow — Budget 30	89
Table 19 – Heuristic versus heuristic — Control-flow — Budget 40	90
Table 20 – Heuristic versus heuristic — Control-flow — Budget 50	90
Table 21 – Heuristic versus heuristic - Control-flow - Budget 100	91
Table 22 – Heuristic versus heuristic — Data-flow — Budget 5	91
Table 23 – Heuristic versus heuristic — Data-flow — Budget 10	92
Table 24 – Heuristic versus heuristic — Data-flow — Budget 20	92
Table 25 – Heuristic versus heuristic — Data-flow — Budget 30	93
Table 26 – Heuristic versus heuristic — Data-flow — Budget 40	93
Table 27 – Heuristic versus heuristic — Data-flow — Budget 50	93
Table 28 – Heuristic versus heuristic — Data-flow — Budget 100	94

Contents

1	Introduction	14
1.1	Motivation	14
1.2	Justification	18
1.3	Objectives	19
1.4	Key findings	20
1.5	Organization	21
2	Background	22
2.1	Defects, infections, and failures	22
2.1.1	Defects	22
2.1.2	Infection	23
2.1.3	Failure	24
2.2	Code coverage	24
2.2.1	Control-flow coverage	25
2.2.2	Data-flow coverage	26
2.2.3	Slicing	28
2.3	Spectrum-based Fault Localization	30
2.4	Final remarks	33
3	Literature review	34
3.1	Methodology	34
3.1.1	Planning	34
3.1.1.1	Research question	34
3.1.1.2	Source selection	35
3.1.1.3	Studies type	35
3.1.1.4	Studies idiom	35
3.1.1.5	Keywords and search string	35
3.1.1.6	Source list	36
3.1.1.7	Inclusion and Exclusion Criteria	36
3.1.2	Conduction	37
3.2	Results	38

3.3	Discussion	42
3.3.1	Programming Languages	42
3.3.2	Validation Setup	43
3.3.3	Max LOC	44
3.3.4	Overhead	44
3.3.5	Data-flow approaches	45
3.4	Conclusion	50
3.5	Final remarks	50
4	Jaguar	51
4.1	Jaguar architecture	51
4.1.1	Invoking test cases and collecting coverage	51
4.1.2	Storing and calculating	53
4.1.3	Results	55
4.1.3.1	Roadmap	55
4.1.3.2	Hierarchical	56
4.2	Final remarks	57
5	Experimental Assessment	58
5.1	Experiment design	58
5.1.1	Research questions	58
5.1.2	Procedure	59
5.1.2.1	Data collection	62
5.1.2.2	Bug localization	62
5.1.2.3	Budgets	63
5.1.3	Statistical Analysis	64
5.2	Results	64
5.2.1	Control- and data-flow effectiveness: barplots	64
5.2.2	Control- and data-flow: statistical tests	65
5.2.3	Heuristic versus Heuristic	66
5.2.4	Efficiency	69
5.3	Discussion	70
5.4	Threats to validity	73

5.5	Final remarks	74
6	Conclusions	75
6.1	Summary	75
6.2	Contributions	76
6.3	Future work	77
	Bibliography	78
	APPENDIX A—Research Strings	83
	APPENDIX B—Heuristic versus heuristics: sta- tistical tests for control- and data- flow coverages	88
B.1	Heuristic versus heuristic: Control-flow	88
B.2	Heuristic versus heuristic: Data-flow	91

1 Introduction

The development of software has to follow the speed of business changes. The Internet brought companies to a world where the requirement of today might be no longer a demand tomorrow. Some companies work in a perpetual development mode, in which the software is never finished and new features are created, evaluated and dismissed every week. Facebook reported that its engineers commit code up to 500 times a day, changing about 3,000 files (FEITELSON; FRACHTENBERG; BECK, 2013). Such a dynamic environment requires tools and methods to make sure that the final product is stable and has less bugs as possible. Testing and debugging are key tasks during the development cycle, which aims to ensure that the software is working as it was designed for. However, they are among the most expensive activities during the development process (CHAIM; MALDONADO; JINO, 2003). Debugging consists of localizing and fixing a program's bug or fault. These activities are accomplished with the help of static information, such as the source code and the bug report, and dynamic information, such as print statements, runtime variable states and test results. Nevertheless, the developer may spend a long time trying to understand and to localize a bug, affecting considerably the overall cost and quality of the software. This is so because fault localization is in general a tedious, time-consuming and error-prone manual debugging task (MAO et al., 2014a; DANDAN et al., 2014). To improve the productivity of developers during the debugging process, various fault localization techniques have been proposed.

1.1 Motivation

Debugging has been studied mainly in two ways. The first one concerns the understanding of the process that a developer utilizes to debug a program. The goal is to analyze the developer's behavior and to understand the cognition models that represent the developer's navigation while debugging. The second way to study debugging is by proposing techniques that support the process utilized by developers to understand the software and to localize bugs more efficiently.

Theories aiming to describe the developer's behavior have been proposed to understand and make predictions about the use of software engineering tools. The result of such

studies are used to guide new software engineering practices and inspire the development of new features for Integrated Development Environment (IDE).

Early theories of program debugging are based on mental models and hypotheses, assuming that the developer reads the program and the bug report to create hypotheses until a fix is found. These theories were mostly developed when IDEs were relatively simple (if an IDE was used at all). Modern IDEs have numerous features such as tool-tips, variable inspection, highlights, clickable links and other aids. Hence, later theories advocate that the developer gathers and organizes the information presented during the debugging process instead of making hypotheses all the time.

Hypotheses creation theory proposes a top-down approach, in which a hierarchy of hypotheses drives the developer towards the understanding of the program (ARAKI; FURUKAWA; CHENG, 1991). The developer starts by making high-level hypothesis, which are a general notion about the code structure and the program domain. The pursuit of these high-level hypothesis leads to more specific questions about inner aspects of the program. Then, low-level hypothesis are made to target the bug fix (LAWRANCE; BOGART, 2013).

The hypotheses are generally just descriptions of functions performed by a component so that the developers do not give them a name. The first hypotheses are global and nonspecific; they concern the overall meaning of the program's components and are usually hard to endorse without further inspection. Therefore, the construction of subsidiary hypotheses are necessary. The most concrete hypothesis are made by the identification of beacons. Beacons are sets of features that may point to tricky structures or operations, like a variable swap operation during a sort algorithm.

Information Foraging Theory is presented by Lawrance e Bogart (2013) as a new way to analyze the developer behavior during the debugging process. It is based on optimal foraging theory, which is about how predators and prey behave during hunting. "Predators sniff for the prey, and follow the scent to the patch where the prey is likely to be" (LAWRANCE; BOGART, 2013, p. 198), trying to save energy and accomplish the goal. Analogously, the developer look for cues and hints to find the path on the code where the bug is likely to be.

The original information foraging constructs are adapted to the debugging world as follows: *Predator* is the developer; *Prey* are the changes necessary to fix the bug, but can also be any information needed to achieve the main goal; *Information patches* are pieces

of the source code and related documents that may contain the prey; *Proximal cues* are words, objects, links and perceptible runtime behaviors in the programming environment; *Information scent* is the perceived likelihood of a cue leading to the prey, it is a measure that exists only in the developer's head; and *Topology* are the paths through the source code and related documents that the developer can navigate.

The experiment conducted by these researchers suggests that information foraging theory presents more data to be analyzed and consequentially reveals more about the behavior of the developer during the navigation. It does not mean that developers do not make hypotheses during the debugging task; they do but not as often as they make use of scents.

Besides analyzing the developer behavior, many techniques have been developed to aid the developer to localize faults. The most common technique is to *print* data useful for debugging purposes during the execution of the program, either on the console or in a logging file. The aim is to record events, such as a piece of code executed or the content of a variable, to help the developer understand the state of the program. This technique is present in most languages and does not require an Integrated Development Environment (IDE) (DELAMARO; CHAIM; VINCENZI, 2010).

Another technique, known as symbolic debugging, allows the developer to issue commands to visualize the content of variables, control the execution of the code and even modify the content of variables. Symbolic debuggers usually offer many features to help the developer understand the state of the program in a specific point (i.e., *breakpoints*), navigate the source code as the programs is executed (i.e., *step-wise navigation*), alter the content of a variable and call specific functions (STALLMAN; PESCH, 1992).

Slicing is a technique used to isolate statements that may affect (or may be affected by) the value of one or more variables directly or indirectly at a particular point of the program or of the execution (WEISER, 1981; KOREL; LASKI, 1988). To find the statements that influence the value of a particular variable, all statements referencing backwards it, directly or indirectly, are part of the slice (data-dependency). Moreover, statements conditionally enabling the execution of other statements that influence the value of the variable in question are also included in the program slice (control-dependency). On the other hand, if the target is to find which statements are affected by a particular variable, the references to this variable are tracked forwards recursively until all the statements affected are considered. Both sides can be analyzed statically or dynamically, static slices

only analyze the source code, with no regard to run-time information. Dynamic slices are based on the run-time information of a particular program execution, hence, only executed statements are inspected.

Spectrum-based fault localization (SFL), also known as coverage-based fault localization, techniques use data collected during a test suite execution to infer which elements of the source code (statements, basic blocks, branches and duas) are more likely of containing the fault (JONES; HARROLD; STASKO, 2002; SANTELICES; JONES; HARROLD, 2009; MAO et al., 2014a)¹. Each element represents a distinct information from the source code. Statements are the lines of code (LOC), basic blocks (or simply blocks) are a set of statements that are always executed in sequence with a single-entry and single-exit point, branches consist of possible transfers of control from one block to another block (such as in if, while and switch commands) and duas represent definition-use associations of variables (RAPPS; WEYUKER, 1985). To determine the elements' likelihood of containing the fault, the source code is firstly instrumented (i.e., the original source code is modified to include code that monitors which element is executed at run-time). Besides the executed code elements (e.g., statements), the test cases results (e.g., fail, success) are also recorded to calculate the suspicious value of each element. This calculation is made in such a way that elements more often executed in failing test cases have a bigger suspicious value than those elements more often executed in passing test cases.

SFL is a promising debugging technique because it identifies excerpts of code with high likelihood of containing bugs and has a relatively low cost at run-time. Most SFL techniques use control-flow coverage, more specifically, statement and block coverage, due to the low cost to collect this data. Though control-flow coverage is helpful to support fault localization, data-flow coverage has been reported as more effective (SANTELICES; JONES; HARROLD, 2009). SFL techniques based on data-flow information make use of *definition-use associations* (dua) to identify suspicious pieces of code. A *definition* occurs in every assignment of value to a variable and a *use* in every reference to a variable's value. A dua consists of a triple, $\langle i, j, x \rangle$, in which the variable x is defined in block i , is used in block j , and there is at least one path between i and j in which x is not modified.

¹ Henceforth, we use indistinctly the terms *spectrum*, *spectra* and *coverage*

1.2 Justification

SFL techniques use information of test runs to evaluate the suspiciousness of program elements (e.g., blocks, branches, duas). These elements are prioritized based on heuristics that establish those more suspicious of containing bugs. The idea is to help developers locate the bugs by examining the suspicious code from higher to lower priority. Test cases had already been created to verify whether the program’s behavior is correct; thus, they can also be used to find the defect that are causing test cases to fail. Until recently, only control-flow coverage, such as statement, block and branch coverage, could be collected at a relatively low overhead.

On the other hand, debugging techniques based on the use of data-flow information have been studied before. DRT ([CHAIM; MALDONADO; JINO, 2003](#)) ranks the most error-revealing definition-use associations (dua) and provide commands to navigate through the test requirements. Techniques to reduce the slice size and increase the chances of hitting the faulty instruction have been recently proposed ([MAO et al., 2014a](#)). Although data-flow information has been shown to perform better than statements and branches to locate the bug site ([SANTELICES; JONES; HARROLD, 2009](#)), the high overhead to collect such an information has prevented its use on industry-level code. Statements and branches can be monitored with 9%-18% runtime overhead while duas have a run-time overhead of 66%-127% ([SANTELICES; JONES; HARROLD, 2009](#)).

Recently, a data-flow coverage monitoring tool, called BA-DUA (Bitwise Algorithm-powered Definition-Use Associations Coverage), was implemented presenting in average 38% runtime overhead for large programs ([ARAUJO; CHAIM, 2014](#)). Such a fairly modest overhead motivates the study of SFL using data-flow information in programs similar to those developed in the industry.

The main hypothesis of this research is that data-flow effectiveness may be due to the greater number of duas in comparison with the number of blocks and branches that are tracked during the test suite. Thereby, the possibility of correlating critical elements with failing test runs are higher when more relations are considered. The goal of this research is to assess this hypothesis.

To achieve such a goal, a comparison between control- and data-flow SFL techniques is carried out. We compare the different techniques using a tool developed for this work, called Jaguar (*JAVa coveraGe faUlt locAlization Ranking*) . Jaguar implements SFL

techniques based on control- and data-flow coverage. It was developed using two coverage tools: JaCoCo², a popular control-flow coverage tool at industrial settings; and BA-DUA. Both tools collect efficiently control- and data-flow coverages. In this sense, Jaguar was designed to be efficient in collecting coverage data.

Differently from previous works, we assess both techniques using open-source programs that are comparable to those developed in the industry. Additionally, we investigate the relation between a coverage (control- or data-flow) and the best known heuristics used in SFL techniques and assess which coverage is more effective; that is, locates more bugs in limited number of blocks. Finally, we compare the costs of SFL based on control- and data-flow coverages. The following research questions summarize the problems addressed in this research:

1. Which heuristic is more effective to support an SFL technique based on control-flow coverage?
2. Which heuristic is more effective to support SFL technique based on data-flow coverage?
3. What coverage type locates more bugs: control- or data-flow coverage?
4. What coverage type ranks the bugs better: control-flow or data-flow coverage?
5. What is the costs associated with the use of control- and data-flow coverages in SFL?

1.3 Objectives

The objective of this work is to analyze and compare the use of control- and data-flow test information in fault localization. To accomplish this goal the following specific objectives are defined:

- to develop an environment to apply the control- and data-flow coverage in fault localization;
- to embed this environment as a plug-in into a well established Integrated Development Environment (IDE) such as Eclipse ³;

² [http://www.eclemma.org/jacoco/.](http://www.eclemma.org/jacoco/)

³ <http://eclipse.org>.

- to perform experiments using benchmarks available in the literature and production-level programs to evaluate the fault localization ability of control- and data-flow coverages;
- to carry out statistical tests to verify whether particular heuristics improve the effectiveness of control- or data-flow coverage and to verify which coverage is more effective for fault localization;
- to assess the costs associated with the use of control- and data-flow in fault localization.

The results of this research contributes to the body of evidence regarding the use of control- and data-flow information in fault localization. They subsidize a practitioner's choice with respect to structural coverage to support his/her testing and debugging activities.

1.4 Key findings

We assessed effectiveness and efficiency of control- and data-flow coverage using 173 faulty versions (real and seeded defects) with projects with sizes varying from 10 to 96 thousand lines of code (KLOC) for 10 heuristics.

Our results indicates that the behavior of the heuristics are similar both to control- and data-flow coverage. Kulczynski2 and Mccon performed better for small number of lines investigated (from 5 to 30 lines) while Ochiai performs better when more lines are inspected (30 to 100 lines).

Moreover, data-flow coverage locates more defects in the range of 10 to 50 inspected lines, being up to 22% more effective. In the range of 20 and 100 lines, data-flow ranks the bug better than control-flow with statistical significance.

Data-flow is more expensive than control-flow: it takes from 23% to 245% longer to obtain the results, on average 129%.

1.5 Organization

This chapter presented the context, motivation, justification, objectives and key findings of our research whose main objective is to compare the effectiveness and efficiency of control-flow and data-flow information for fault localization.

The remainder of this dissertation is organized as follows:

- Chapter 2 presents concepts about defects, infections, failures, control-flow, data-flow, slicing and spectrum-based fault localization.
- Chapter 3 examines the related work by conducting a systematic research.
- Chapter 4 presents Jaguar — a new software for coverage-based fault localization using control- and data-flow information.
- Chapter 5 describes the experiment with Jaguar and selected programs, the results and their discussion.
- Finally, Chapter 6 contains the conclusions drawn.

2 Background

This chapter presents the main concepts utilized in this research. We start off by defining the concepts of defect, infection, and failure. Since the focus of this research is on coverage-based debugging, we present the different types of code coverage that are used for debugging purposes. Moreover, we discuss the concept of program slicing due to the similarity with the data-flow coverage utilized in this proposal. We conclude the chapter with the presentation of the main concepts regarding coverage-based debugging.

2.1 Defects, infections, and failures

Each author has different definitions for basic debugging terms ([IEEE... , 1990](#)) ([HUIZINGA; KOLAWA, 2007](#)). We will use in this document the terminology presented by Zeller ([ZELLER, 2005](#)).

2.1.1 Defects

A defect — also known as fault or bug — is an incorrect piece of code that can cause an infection. The defect can be caused by the developer’s lack of knowledge about the requirements or technology, a program state not predicted by the original requirements, incompatible interfaces between two modules, or an unpredictable interaction of several components.

Figure 1 shows the code of a simple function, named `max`, obtained from ([CHAIM; ARAUJO, 2013](#)). It receives two parameters: the first is an `int array` and the second is the array size. The function is supposed to return the array biggest number, but there is a fault at line 4. The first three columns represent line, statement and node numbers, respectively. Only lines that contain instructions are presented in Figure 1. A node is a set of instructions executed in such a way that once the first one is executed all are executed in sequence.

In line 4, the command `array[++i]` should be `array[i++]`; that is, the increment (`++`) must come *after* the variable `i`. This causes variable `max` to be assigned to the second position of the array, because `i` starts as 0 and is increased by 1 *before* being used as the

array element position. This defect will be executed every time the function is called, since it is in the first node.

A defect can be reached during the execution of a test case, but it does not always causes an infection. Some defects will only trigger an infection if particular conditions are fulfilled.

Figure 1 – Code of `max` program

Line	Statement	Node	Code
1	-	-	<code>int max(int[] array, int length)</code>
2	-	1	<code>{</code>
3	1	1	<code>int i = 0;</code>
4	2	1	<code>int max = array[++i]; //array[i++];</code>
5	3	2	<code>while(i < length)</code>
6	-	3	<code>{</code>
7	4	3	<code>if(array[i] > max)</code>
8	5	4	<code>max = array[i];</code>
9	6	5	<code>i++;</code>
10	-	5	<code>}</code>
11	7	6	<code>return max;</code>
12	-	6	<code>}</code>

Source: [Chaim e Araujo \(2013\)](#)

2.1.2 Infection

An infection (or error) is detected when the program state is not as it was supposed to be. The defect was executed under such conditions that trigger an infection. One infection can cause more infections by passing an unexpected state to pieces of code with no defects.

In the previous example, the `max` function, the infection is triggered when line 4 is executed. The variable `max` holds the value of the array's second element, instead of the value of the first element. At this point, the program is in a state that is not correct. The variable `max` should be holding the array's first element to iterate through it and find the biggest value. Nevertheless, if the biggest value is not in the first array element, the infection will be healed since the biggest value will eventually be found by the iteration starting from the second element.

Therefore, once there is an infection, a failure may occur. Likewise the defect, the infection can exist, but that does not guarantee that the user will observe a failure.

2.1.3 Failure

Failure is an external observable infection or error. The infection propagates and then generates an unexpected behavior of the program. The failure is visible to the end user, like an error message or wrong outputs.

The `max` function has a defect that triggers an infection every time the defect is executed, but it does not always generate a failure. Only two cases will make the program fail. The first case is when the array has only one element. When executing line 4 the program will generate an exception due to the attempt to access the array's second element. The second case is when the biggest element is in the first position. The first element will be "missed" due to the defect, which initializes the variable `max` considering the second element. As a result, the array will be iterated from the second to the last element. The first case shows an error message and the second case results in wrong outputs.

As stated by Dijkstra, testing can only show the presence of defects, but never their absence (ZELLER, 2005). If the defect exists but never generates a failure, all the test cases will pass. That is one of the reasons why test coverage is been used as a measure of the quality of a test suite. With higher coverage, the chances of a defect not being detected are lower.

2.2 Code coverage

Coverage data are information indicating which software components were executed by a specific run. Different components can be monitored such as statements (YOU et al., 2013), nodes, slices (MAO et al., 2014a), data-dependences (CHAIM; MALDONADO; JINO, 2003), and control-dependences (DANDAN et al., 2014).

The program needs to be instrumented to collect coverage information during the execution (ARAUJO; CHAIM, 2014). The instrumentation consists of extra code to track each component, recording whether it was executed or not. The run-time information is collected during a test suite execution.

2.2.1 Control-flow coverage

Statements are lines of code that contains instructions. As can be noticed in Figure 1, there are 12 lines, 7 statements and 6 nodes. The first two lines does not count as statements because they do not have instructions and consequently do not alter the state of the program. Nevertheless, the assignment of values to formal parameters occurs in the first statement which it is located at line 1.

Control-flow information of a program is represented by a *graph* with *nodes* and *edges*. Each *node*, also referred to as *block*, represent a set of statements that are always executed in sequence, implying that once the first statement is executed all statements in the node will be executed. The *edge*, also referred to as *branch*, represents the transfer of control from one node to another node due to conditional (e.g., *if*, *switch*, *for* and *while*) or unconditional transfer commands (e.g., *goto*, *break*, and *continue*) (HECHT, 1977).

Figure 2 – Control-flow graph of *max* program

Source: Chaim e Araujo (2013)

Figure 2 shows the Control-flow Graph of *max* program. The node 2, for example, represents the statement at line 5 that contains the *while* command. From this point, the program execution can be directed to two distinct nodes. If the condition of the *while* command is *true* the node 3 is executed; otherwise, the node 6 is called.

Table 1 specifies all nodes of the *max* program. As detailed in Figure 1, they start at 1 and go until 6. Table 1 also specifies all the possible edges extracted from the *max* program. These edges represent all the arrows from Figure 2, originating from one node and directing to another.

Table 1 – All nodes and all edges of *max* program.

All nodes	All edges
1	(1,2)
2	(2,3)
3	(2,6)
4	(3,4)
5	(3,5)
6	(4,5)
	(5,2)

Source: Souza (2012)

Let N be the set of nodes of a program G such that every node n belongs to N and E the set of edges (n', n) , such that $n' \neq n$, which represents a possible transfer of control between node n' and node n . A path is a sequence of nodes $(n_i, \dots, n_k, n_{k+1}, \dots, n_j)$, where $i \leq k < j$, such that $(n_k, n_{k+1}) \in E$ (CHAIM; ARAUJO, 2013).

A node (edge) is considered covered if there is a test case that traverses a path that includes such a node (edge). Two testing criteria, *all-nodes* and *all-edges*, require that every node and every edge of a program, respectively, be covered by at least one test case to be satisfied.

Coverage information of nodes and edges obtained from the execution of test suites can be used to infer the bug localization.

2.2.2 Data-flow coverage

Data-flow information focuses on variables definitions and uses. A *definition* of a variable happens when it receives a new value. It might occur either when the variable is initialized or when its value is changed. A *use* of variable occurs when it is referred to. This *use* can happen in two ways. The first one is to compute a value, as at line 8 of the Figure 1 (`max = array[i];`), in which variables `array` and `i` are used to compute the value of variable `max`. The second way of using a variable is to compute a predicate, as in line 5 (`while(i < length)`), variables `i` and `length` are used to decide which path to follow. The former is called *computational use* (*c-use*) and the latter *predicative use* (*p-use*).

Figure 3 shows the data-flow information of each node and edge of the control-flow graph. The first node, for instance, holds the *definition* of four distinct variables (`i`, `array`,

Figure 3 – Control-flow graph of the max program including data-flow information

Source: [Chaim e Araujo \(2013\)](#)

`length` and `max`). The *p-uses* of variables `i` and `length` at line 5, described earlier, are associated with edges (2,3) and (2,6). The *c-use* of variable `array` at line 8, described earlier, is associated with node 4, along with the *c-use* of variable `i`.

A *definition-clear path* with respect to a variable `x` is a path $(n_i, \dots, n_k, n_{k+1}, \dots, n_j)$, where $i \leq k < j$, such that `x` is not redefined, except possibly in the last node.

A *definition-use association* (dua) $\langle i, j, x \rangle$ represents a data-flow requirement in which a definition of variable `x` occurs in node `i` and a *c-use* occurs in node `j`, and there is a *definition-clear path* with respect to `x` from `i` to `j`.

Likewise the triple $\langle i, (j, k), x \rangle$ represents a data flow requirement in which a definition of `x` occurs in node `i` and a *p-use* in edge (j,k). Additionally, there is a path $(i, \dots, j, k)$ that is definition-clear with respect to `x`.

Considering only *c-uses*, variable `max` in program `max` has two duas ($\langle 1, 6, \text{max} \rangle$, $\langle 4, 6, \text{max} \rangle$). The first dua, $\langle 1, 6, \text{max} \rangle$, means that variable `max` is defined at node 1 and used at node 6. This dua can only be considered as covered if, during the test execution, the variable used at node 6 was not modified after its definition at node 1, in other words, has a definition-clear path. If `max` is redefined at node 4, and then used at node 6, the dua $\langle 4, 6, \text{max} \rangle$ is considered as covered instead.

Considering *p-uses*, variable `max` has four duas ($\langle 1, (3,4), \text{max} \rangle$, $\langle 1, (3,5), \text{max} \rangle$, $\langle 4, (3,4), \text{max} \rangle$, $\langle 4, (3,5), \text{max} \rangle$). The first dua, $\langle 1, (3,4), \text{max} \rangle$, means that variable `max` is defined at node 1, is used as a predicate at node 3 and directs the execution to node 4. When the `if` condition, in node 3, is true, the execution goes towards node 4, thus, this dua is covered.

Thereafter, the variable *max* has its value modified at node 4, by the command `max = array[i];`. Thus, a new definition of the variable takes place. If node 3 is executed again and no redefinition of *max* occurs, one of the following duas will be covered: $\langle 4, (3,4), \text{max} \rangle$ or $\langle 4, (3,5), \text{max} \rangle$. In both, the definition is made at node 4 (`max = array[i];`), and the predicate use starts at node 3 (`array[i] > max`). If the result of the command `array[i] > max` is true, node 4 will be executed, hence, $\langle 4, (3,4), \text{max} \rangle$ is considered as covered, otherwise, node 5 is executed, and $\langle 4, (3,5), \text{max} \rangle$ is considered as covered.

Table 2 – All definition-use associations of the *max* program.

All uses			
(1, 6, max)	(1, 4, i)	(5, 4, i)	(1, 4, array)
(4, 6, max)	(1, 5, i)	(5, 5, i)	(1, (3,4), array)
(1, (3,4), max)	(1, (2,3), i)	(5, (2,3), i)	(1, (3,5), array)
(1, (3,5), max)	(1, (2,6), i)	(5, (2,6), i)	(1, (2,3), length)
(4, (3,4), max)	(1, (3,4), i)	(5, (3,4), i)	(1, (2,6), length)
(4, (3,5), max)	(1, (3,5), i)	(5, (3,5), i)	

Source: [Chaim e Araujo \(2013\)](#)

Table 2 specifies all the definition-use associations (dua) of *max* program. Thus, it contains all the possible ways that a variable can be defined and used in this program. A test case covers a subset of them, but hardly all of them. Data-flow information is expensive to monitor due to the amount of duas that a program can have. For instance, the *max* program has 12 lines (7 statements) and 23 duas. Therefore, the number of duas is usually bigger than the number of lines of code.

The all-uses criterion ([RAPPS; WEYUKER, 1985](#)) establishes that a test set to satisfy it should include at least one test case that covers every dua of the program. A test set covers a dua (i, j, x) or $(i, (j,k), x)$ if it traverses a definition clear path $(i,...,j)$ or $(i,...,j,k)$ with respect to *x*.

2.2.3 Slicing

Data dependency between two variables happens when variable v_1 influences the value of another variable v_2 . On the previous example, at line 8, *max* has a data dependency with `array[i]` because it will receive the value of that variable. Control dependency between two variables happens when a variable v_1 is conditionally guarded by another

variable v_2 . On the previous example, the variable `array[i]`, at line 7, has a control dependency on the variable `length`, at line 5. Depending on the value of `length` the next line might be executed or not.

Slicing is a technique used to isolate statements that directly or indirectly might affect the value of one or more variables at a particular point of a program or of its execution (JU et al., 2014a). In order to find the statements that influence the value of a particular variable several approaches have been devised. Some of them are presented as follows:

- **Static backward slice (SBS)**: it includes all statements that can influence the value of a variable, taking into account all possible paths. Because it is static, the analysis is carried out only by looking at the code; that is, there is no need to execute the program (MAO et al., 2014a).
- **Dynamic backward slice (DBS)**: it includes statements that influence the value of a variable, during the execution of a particular test case. Because it is dynamic, the analysis is performed at run-time. Different executions can generate different slices of the same variable, because the state of the program can differ for different test cases (MAO et al., 2014a).
- **Execution slice (ES)**: it includes all statements that were executed during an execution of a test case. This approach includes in the slice even statements that has no data or control dependency with respect to the output variable (JU et al., 2014a).

Figure 4 – Slices of variable `max` at line 11 when running `max([4,3,2],3)`

Line	Statement	Node	Code	SBS	DBS	ES
1	-	-	<code>int max(int[] array, int length)</code>	•	•	•
2	-	1	<code>{</code>			
3	1	1	<code>int i = 0;</code>	•	•	•
4	2	1	<code>int max = array[++i]; //array[i++]</code>	•	•	•
5	3	2	<code>while(i < length)</code>	•		•
6	-	3	<code>{</code>			
7	4	3	<code>if(array[i] > max)</code>	•		•
8	5	4	<code>max = array[i];</code>	•		
9	6	5	<code>i++;</code>	•		•
10	-	5	<code>}</code>			
11	7	6	<code>return max;</code>	•	•	•
12	-	6	<code>}</code>			

Figure 4 present the code of the `max` program along with the three slices presented before. The last three columns represent, respectively, the Static Backward Slice (SBC), Dynamic Backward Slice (DBS) and Execution Slice (ES). For the dynamic slices (DBS and ES) it is used a test case with the parameters `array = [4,3,2]` and `length = 3`.

Due to the low complexity of the example, a static backward slice of the `max` variable at line 11 would include all the statements, as showed in Figure 4. The `max` variable is data dependent to the variables `array` and `i`, as can be seen at line 8, which includes all statements that change those variables. Besides the data-dependency, all control dependent statements, which includes lines 5 and 7, must be added to the static backward slice statements set.

For a dynamic backward slice of the same variable `max` at line 11, the slice would include only lines 1, 3 and 4. Line 4 changes the value of `max` and includes `array` and `i` as data dependent, hence, line 1 is included because it is where happens the definition of `array` and line 3 is also included because it is where happens the definition of `i`. The remanding statements are not included mainly because line 8 is never executed in this run. As `max` variable receives the value of the second element of `array`, which is 5, it will never pass the condition at line 7. When a different input is used, different statements will be executed, changing the dynamic slice.

The execution slice include all lines, except line 8. This line is not executed because `max` is initialized, erroneously, with the second element of the array (5) and then the condition at line 7 is never satisfied.

Because it considers all possible paths, SBC is usually large, affecting its effectiveness. DBS analyzes only one execution, narrowing down the size of the result, but still with a fine accuracy (MAO et al., 2014a). Although ES is dynamic, it generates slices too large to guide the developers in locating faults effectively (JU et al., 2014a).

2.3 Spectrum-based Fault Localization

Spectrum-based Fault Localization (SFL), also known as Coverage-based Fault Localization (CBFL), is a technique that uses the program's run-time information to find the most likely peaces of code that contain the fault. Besides the components (node, edge or dua) executed during each test, SFL needs to save the test result (pass or fail). Then, these data are used to define the suspiciousness of each component. This value is calculated using

one of the many heuristics presented in the literature (JONES; HARROLD; STASKO, 2002) (MAO et al., 2014a) (JU et al., 2014a). Regardless of the chosen heuristic, all of them assume the following principles:

- The more a component is executed by passing test cases, the less suspicious it will be.
- The more a component is not executed by passing test cases, the more suspicious it will be.
- The more a component is executed by failing test cases, the more suspicious it will be.
- The more a component is not executed by failing test cases, the less suspicious it will be.

Hence, even when a component is not executed its suspiciousness is affected. Components not executed by failed test cases are less likely to contain the defect than components not executed by passed test cases.

As Table 3 summarizes, each component j has four coefficients, $c_{ef}(j)$, $c_{ep}(j)$, $c_{nf}(j)$ e $c_{np}(j)$. The $c_{ef}(j)$ represents the number of *failed* test cases that *executed* the component j , $c_{ep}(j)$ represents the number of *passed* test cases that *executed* the component j , $c_{nf}(j)$ represents the number of *failed* test cases that did *not execute* j and $c_{np}(j)$ represents the number of *passed* test cases that did *not execute* j .

Table 3 – SFL Coefficients

	Failed Test	Passed Test
Executed j	$c_{ef}(j)$	$c_{ep}(j)$
Not Executed j	$c_{nf}(j)$	$c_{np}(j)$

Source: Henrique Ribeiro, 2016

SFL techniques use heuristics to calculate the components suspiciousness. Many heuristics have been studied by different authors, 16 of them are listed by Mao et al. (MAO et al., 2014a). We present in Table 4 10 heuristics utilized in SFL.

One of the first heuristic for fault localization proposed was Tarantula (JONES; HARROLD; STASKO, 2002) whose formula (H_T) is shown in Table 4 (first row and first column). It determines a suspicious value for each component j using the coefficients described in Table 3. The suspiciousness value of the components are ranked in descending order so that the most suspicious components are the first to be examined.

Table 4 – Heuristics for fault localization

Heuristic	Formula
Tarantula	$\frac{\frac{c_{ef}}{c_{ef}+c_{nf}}}{\frac{c_{ef}}{c_{ef}+c_{nf}} + \frac{c_{ep}}{c_{ep}+c_{np}}}$
Ochiai	$\frac{c_{ef}}{\sqrt{(c_{ef}+c_{nf})(c_{ef}+c_{ep})}}$
Jaccard	$\frac{c_{ef}}{c_{ef}+c_{nf}+c_{ep}}$
Zoltar	$\frac{c_{ef}}{c_{ef}+c_{nf}+c_{ep}+10000 \cdot \frac{c_{nf}c_{ep}}{c_{ef}}}$
O^p	$c_{ef} - \frac{c_{ep}}{c_{ep}+c_{np}+1}$
Minus	$\frac{\frac{c_{ef}}{c_{ef}+c_{nf}}}{\frac{c_{ef}}{c_{ef}+c_{nf}} + \frac{c_{ep}}{c_{ep}+c_{np}}} - \frac{1 - \frac{c_{ef}}{c_{ef}+c_{nf}}}{1 - \frac{c_{ef}}{c_{ef}+c_{nf}} + 1 - \frac{c_{ep}}{c_{ep}+c_{np}}}$
Kulczynski2	$\frac{1}{2} \left(\frac{c_{ef}}{c_{ef}+c_{nf}} + \frac{c_{ef}}{c_{ef}+c_{ep}} \right)$
McCon	$\frac{c_{ef}^2 - c_{nf}c_{ep}}{(c_{ef}+c_{nf})(c_{ef}+c_{ep})}$
Wong3	$c_{ef} - p$, where $p = \begin{cases} c_{ep} & \text{if } c_{ep} \leq 2 \\ 2 + 0.1(c_{ep} - 2) & \text{if } 2 < c_{ep} \leq 10 \\ 2.8 + 0.001(c_{ep} - 10) & \text{if } c_{ep} > 10 \end{cases}$
DRT	$\frac{c_{ef}}{1 + \frac{c_{ep}}{ T }}$ where $ T $ is the size of test suite T

Source: Henrique Ribeiro, 2016

Figure 5 presents the coverage information of the max program. The first three columns are equivalent to the columns of Figure 1. The next five columns represent the coverage of each test from the test suite detailed in Table 5. The bullet symbol ($\bullet$) means that the line was covered by the test and its absence means that the line was not covered. The following columns regard the four coefficient explained before, and the last column is the suspiciousness value calculated using the Tarantula formula.

Figure 5 – Coverage of max function with Tarantula heuristic

Line	Statement	Node	t_1	t_2	t_3	t_4	t_5	c_{np}	c_{ep}	c_{nf}	c_{ef}	H_T
1	-	-	$\bullet$	$\bullet$	$\bullet$	$\bullet$	$\bullet$	0	3	0	2	0.5
2	-	1	$\bullet$	$\bullet$	$\bullet$	$\bullet$	$\bullet$	0	3	0	2	0.5
3	1	1	$\bullet$	$\bullet$	$\bullet$	$\bullet$	$\bullet$	0	3	0	2	0.5
4	2	1	$\bullet$	$\bullet$	$\bullet$	$\bullet$	$\bullet$	0	3	0	2	0.5
5	3	2	$\bullet$	$\bullet$	$\bullet$	$\bullet$		0	3	1	1	0.33
6	-	3	$\bullet$	$\bullet$	$\bullet$	$\bullet$		0	3	1	1	0.33
7	4	3	$\bullet$	$\bullet$	$\bullet$	$\bullet$		0	3	1	1	0.33
8	5	4	$\bullet$	$\bullet$	$\bullet$			0	3	2	0	0
9	6	5	$\bullet$	$\bullet$	$\bullet$	$\bullet$		0	3	1	1	0.33
10	-	5	$\bullet$	$\bullet$	$\bullet$	$\bullet$		0	3	1	1	0.33
11	7	6	$\bullet$	$\bullet$	$\bullet$	$\bullet$		0	3	1	1	0.33
12	-	6	$\bullet$	$\bullet$	$\bullet$	$\bullet$		0	3	1	1	0.33
			✓	✓	✓	✗	✗					

Source: Souza (2012)

Table 5 – Test Suite

T_n	Test	Expected Result	Actual Result
t_1	$\max([1,2,3], 3)$	3	3
t_2	$\max([5,5,6], 3)$	6	6
t_3	$\max([2,1,10], 3)$	10	10
t_4	$\max([4,3,2], 3)$	4	3
t_5	$\max([4], 1)$	4	error

Source: Souza (2012)

The line number 5, for instance, was *not executed* by failed test cases ($c_{nf} = 0$), was *executed* by three passed test cases ($c_{ep} = 3$), was *not executed* by one failed test cases ($c_{nf} = 1$) and was *executed* by one failed test cases ($c_{ef} = 1$), then its suspiciousness value using the Tarantula Heuristic is 0.33.

The top four lines have the same coefficients, thereby the same suspiciousness value. The SFL technique based on the Tarantula heuristic would rank these lines as more likely of containing the fault. So the developer is advised to search for the fault firstly in these lines. In this particular case, the fault is located in the most suspicious lines.

Any of the heuristics described in Table 4 could be used to determine the suspiciousness of the statements of the example program. We will examine in Chapter 6 how the heuristics impact the effectiveness of control- and data-flow coverage in fault localization.

2.4 Final remarks

This chapter presented the fundamental concepts related to this work, namely, the concepts of defect, infection and failure (Section 2.1); control- and data-flow coverage (Sections 2.2.1 and 2.2.2); slicing techniques (Section 2.2.3); and spectrum-based fault localization (Section 2.3). A literature review regarding this research is presented next.

3 Literature review

In this chapter, we present a systematic literature review on the use of data-flow coverage in Spectrum-based Fault Localization (SFL). The details of the review, the main results and their discussion are presented next.

3.1 Methodology

Systematic review (SR) is a method to identify, validate, and interpret the relevant research available regarding a specific research question ([KITCHENHAM, 2004](#)). A systematic review differs from a non-systematic review by following a protocol and a sequence of steps previously defined. This approach permits the research to be reproduced and mitigate bias ([BIOLCHINI et al., 2005](#)).

The SR protocol used by this work was based on the directives proposed by [Kitchenham \(2004\)](#) and [Biolchini et al. \(2005\)](#). The procedures for planning, conducting, and extracting the data for this SR are detailed below.

3.1.1 Planning

We conducted an exploratory research in which seminal papers regarding Spectrum-based Fault Localization (SFL) were studied to extract the keywords used in the protocol. Following the guidelines proposed by [Kitchenham \(2004\)](#), the research protocol of this work is presented next.

3.1.1.1 Research question

The objective of the proposed systematic review is to analyze the use of data-flow information in SFL. To address this objective, we defined the following research questions:

1. How has data-flow coverage information been used in SFL?

Regarding the topics of the research question, the following information was defined:

- **Intervention:** approaches and results of fault localization techniques that uses data-flow information.

- **Control:** similar reviews.
- **Population:** publications regarding fault localization based on data-flow information.
- **Results:** analysis of the techniques found during the research, highlighting their strong and weak points.
- **Application:** researchers interested in data-flow spectrum-based techniques and developers studying new ways to improve fault localization.

3.1.1.2 Source selection

Sources should be available on websites, preferably on well known digital libraries of the information technology area. Papers from other sources might be included provided they comply with the systematic review requirements.

3.1.1.3 Studies type

We considered papers published in scientific events and journals that detail fault localization techniques based on data-flow information.

3.1.1.4 Studies idiom

English.

3.1.1.5 Keywords and search string

Two main keywords were identified: “data-flow” and “spectrum-based fault localization”. The search string included words that could represent the use of *data-flow* techniques such as slice and definition-use associations; and also synonymous of *spectrum-based fault localization* such as coverage-based fault localization. Various ways of spelling the same word, as well as abbreviations, were included with the *OR* logical operand. The strings submitted to each database are listed in [Appendix A](#).

3.1.1.6 Source list

1. ACM Digital Library ¹
2. IEEE Xplore Digital Library ²
3. Science Direct ³
4. Wiley Online Library ⁴
5. Scopus ⁵

3.1.1.7 Inclusion and Exclusion Criteria

After submitting the research query string to each of the previous listed database sources, the title and the abstract of every resulted papers were read to verify whether they *fit* all the *inclusion criteria* and *do not fit* any of the **exclusion criteria**. We do not use any criterion based on the publishing date. The inclusion and exclusion criteria are listed bellow:

Inclusion criteria:

1. it will be included studies published and fully available at digital libraries or printed version.
2. it will be included studies which have already been approved by the scientific community ⁶.
3. it will be included studies that utilize data-flow SFL localization techniques.

Exclusion Criteria:

1. it will be excluded studies that do not use data-flow techniques for SFL.
2. it will be excluded studies that do not specify how data-flow information is utilized for fault localization.
3. it will be excluded studies that are not written in one of the accepted languages (Portuguese and English).

¹ <http://dl.acm.org/>

² <http://ieeexplore.ieee.org/>

³ <http://www.sciencedirect.com/>

⁴ <http://onlinelibrary.wiley.com/>

⁵ <https://www.scopus.com/>

⁶ The study should have been published in peer-reviewed journals or conference proceedings, for papers, or by an examination board, for academic works (Master's thesis or Phd's dissertations).

4. it will be excluded studies that present the technique but do not validate it.

Papers not filtered after applying these criteria were then fully read to extract the data needed to complete the systematic review (SR). The next step is the *conduction* in which the presented protocol is applied.

3.1.2 Conduction

Table 6 – Data base research result

Data-base	All	Included	Excluded	Duplicated
ACM	15	4	11	0
IEEE	43	10	29	4
Capes	7	0	7	0
Wiley	45	1	43	1
ScienceDirect	13	3	10	0
Scopus	104	8	33	63
Total	220	26	126	68

Source: Henrique Ribeiro, 2016

The research was conducted during November 2014. Table 6 summarizes the results obtained. It was returned 220 papers in which 68 studies were present in more than one database (duplicated) and 126 articles were excluded from the SR because they did not satisfy all the inclusion criteria and/or satisfied at least one exclusion criteria. Hence, 26 papers were selected to be read in its entirety. Only 11% of all the returned papers were further analyzed by this SR, as can be seen in Figure 6.

Figure 6 – Inclusion and exclusion criteria result

Source: Henrique Ribeiro, 2016

Figure 7 represents the distribution of Included, Excluded, and Duplicated papers throughout each database.

--	--	--	--	--	--	--

Paper	Tool name	Lang.	Heuristic	Programs tested	Faulty versions	Max KLOC
continues in the next page						

Table 7 – continuation

Paper	Tool name	Lang.	Heuristic	Programs tested	Faulty versions	Max KLOC
(CHAIM; MALDONADO; JINO, 2003)	gdb/poke	C	New Heuristic	Sort(unix)	11	1
(MAO et al., 2014b)	SSFL	C	16 Heuristics	Siemens, space, flex, grep and sed	257	10
(SANTELICES et al., 2009)	DUA - FORENSICS	Java	Ochiai	Siemens, NanoXML, XML-security and JABA	107	38
(ALVES et al., 2011)	—	Java	Tarantula	Siemens, Jtopas, Ant	50	25-80
(WEN et al., 2011)	JHSA	Java	Tarantula	JHSA	178	11
(JU et al., 2014b)	HSFal	Java	New Heuristic	Siemens, Jtcas, Sorting, NanoXML and XML-security	104	22
(MASRI, 2010)	DIFA	Java	Tarantula	Jaligner and NanoXML	22	7
(ZHANG et al., 2014)	EMMA + JSLICE	Java	Nash1, Binary, GP02, GP03, GP19	Siemens	71	0.5

continues in the next page

Table 7 – continuation

Paper	Tool name	Lang.	Heuristic	Programs tested	Faulty versions	Max KLOC
(LIU et al., 2013)	—	Java	New Heuristic	Siemens, NanoXML	74	3.5
(MA et al., 2013)	—	C	New Heuristic	Siemens	113	5
(CAO et al., 2014)	DSFL	Java	—	Siemens, NanoXML, XML-security	111	22
(HE et al., 2014)	CPSS	C	Tarantula, CT, SBI	SIR	—	—
(LEI et al., 2012)	SSFL	C	8 Heuristics	Siemens, Space	154	10
(ZHANG; KIM; KHURSHID, 2013)	FaultTracer	Java	Tarantula, Jaccard and Ochiai	Jtopas, xml-security, Jmeter, Ant	23	80
(YANG; WU; LIU, 2012)	—	Java	New Heuristic	XML-security, Jtopas	—	22
(HOFER; WOTAWA, 2012)	Sendys	Java	Ochiai	Bank Account, Mid, Static Eample, Traffic Light, ATMS, Reflec. Visitor, Jtopas, Tcas	42	4
(YU et al., 2011)	—	C	Tarantula	Siemens (replace, printtokens, printtokens2)	18	0.5

continues in the next page

Table 7 – continuation

Paper	Tool name	Lang.	Heuristic	Programs tested	Faulty versions	Max KLOC
(XU et al., 2011)	—	C	Tarantula, Ochiai and Heuristic III	Siemens, gzip, grep, make	207	5
(ASSI; MASRI, 2011)	—	Java	New Heuristic	Siemens (tot_info, replace, tcas)	18	0.5
(EICHINGER et al., 2010)	—	Java	New Heuristic	Weka	16	301
(SUN; LI; NI, 2008)	Dicotomy	C	Tarantula	Siemens	142	0.5
(WANG; ROY-CHOUDHURY, 2007)	—	Java	—	Siemens (schedule, print_tokens)	16	0.5
(SUN et al., 2007)	—	C	—	Tower Simulator System	1000	1
(WONG; QI, 2006)	DESiD	C	—	Space	10	10
(WONG; QI, 2004)	DESiD	C	—	Space	10	10
(AGRAWAL et al., 1995)	chislice (ATAC + xSlice)	C	—	Sort (unix)	25	1

Source: Henrique Ribeiro, 2016

Data-flow techniques were divided in six types for a better understanding on how data-flow is explored over each study. The first, and most common, type of data-flow technique is *program slicing*, used by 12 papers (MAO et al., 2014b), (ALVES et al., 2011), (WEN et al., 2011), (JU et al., 2014b), (ZHANG et al., 2014), (LIU et al., 2013), (HE et

al., 2014), (LEI et al., 2012), (HOFER; WOTAWA, 2012), (YU et al., 2011), (SUN; LI; NI, 2008), (WANG; ROYCHOUDHURY, 2007). *Duas* were used by five studies (CHAIM; MALDONADO; JINO, 2003), (SANTELICES et al., 2009), (ZHANG; KIM; KHURSHID, 2013), (XU et al., 2011), (ASSI; MASRI, 2011). The third type uses operations (union, intersection, subtraction, addition) on slices from different test cases; it is called *program dicing*. It was used in four works (SUN et al., 2007), (WONG; QI, 2006), (WONG; QI, 2004), (AGRAWAL et al., 1995). Two papers (EICHINGER et al., 2010), (MASRI, 2010) exploited the use of method call graph with the addition of data-flow information (e.g., method parameters, return variables); this technique is called *Method call with data-flow*. A fifth type of data-flow technique was introduced in two works (WONG; QI, 2006), (WONG; QI, 2004); it utilizes the data-dependency between two different blocks to improve fault localization, being called here *Block-data-dependency*. Finally, the last type of data-flow technique is used by a single research (YANG; WU; LIU, 2012) and consists of a combination of *dua* and control-flow to elaborate chains of data- and control-flow dependencies. We refer to it as *Data-chain*. This information is summarized in Figure 8.

Figure 8 – Distribution of the type of data-flow techniques over all papers

Source: Henrique Ribeiro, 2016

3.3 Discussion

3.3.1 Programming Languages

One can observe on Table 7 that only two program languages are supported by debugging tools — C and Java. Java is the preferred language, used in fourteen out of twenty six papers, whereas C was utilized in twelve works. While C and Java are widely used by the industry, they are also preferred in the academic realm. As shown in Figure 9, the C language was used by all (except one) studies until 2008. From 2008 on, six

new approaches of SFL using data-flow technique also used the C language, meanwhile, thirteen techniques were implemented using the Java language. So, the trend seems to be that Java will be the most used language by novel debugging approaches.

Figure 9 – Programming languages used by each approach over the years

Source: Henrique Ribeiro, 2016

3.3.2 Validation Setup

Concerning the validation setup, which is the programs and faults used to validate the proposed technique, most studies used programs from the Software-artifact Infrastructure Repository⁷ (SIR), which provides C and Java programs containing faults. Among the SIR programs, the Siemens suite (tcas, schedule, schedule2, totinfo, printtokens, printtokens2, and replace) is the most used benchmark by the studies presented in this systematic review. Space, flex, grep, gzip, and make are also programs provided by SIR and used in some of the validation setups. Some studies used a Java version of the Siemens suite.

The SIR programs was utilized by seventeen of the twenty six studies; the Siemens suite were used by fourteen of them. NanoXML and XML-security was used in five works; Jtopas was used in four studies; Unix, Sort, and Ant were utilized in two works. Some programs were used by a single study (Tower Simulator System, Bank Account, Mid, Static Sample, Traffic Light, ATMS, Reflec. Visitor, JABA, Weka, JHSA, Jmeter, Jtcas, Jalinger and Sorting).

Despite being the most used benchmark in fault localization studies, the Siemens suite does not represent the characteristics of production-level programs. It consists of seven programs with 310 LOC and 3115 test cases each, on average (MAO et al., 2014b).

⁷ <http://sir.unl.edu/portal/index.php>

These are not the type of programs developed at industrial settings, which usually are bigger and include less test cases.

3.3.3 Max LOC

The last column of Table 7, called *Max KLOC*, contains the number of lines of code (LOC) of the biggest program used to validate each technique. This information is highlighted to assess the applicability of the technique in industry-level programs. No more than seven studies validated the technique over programs with more than 12 KLOC. Only Weka has more than 80 KLOC. This is the biggest program used among the twenty six studied in this work. Thus, further research is necessary to investigate the applicability of data-flow approaches for spectrum-based fault localization in programs similar to those developed in the industry.

3.3.4 Overhead

We notice that sixteen papers do not cite overhead information. On the remaining ten studies: two studies compare the overhead with the traditional SFL (JU et al., 2014b; MAO et al., 2014b); one work summarizes the results only for some programs (ALVES et al., 2011); one paper compares itself with other data-flow coverage types (MASRI, 2010); one work cites the computational complexity of the technique to refine the search for duas (CHAIM; MALDONADO; JINO, 2003); one study considers its computational overhead as marginal compared to the basic approaches (HOFER; WOTAWA, 2012); one work cites that the time varies significantly across different subjects but still on average a little slow than a similar approach (ZHANG; KIM; KHURSHID, 2013). Three studies cite that their technique are not efficient (ASSI; MASRI, 2011), with high time complexity (YU et al., 2011) and have low overhead only with small programs (LEI et al., 2012).

Most of the approaches does not cite overhead information and some researches acknowledge that it is expensive to collect the data specially when using large programs. Hence, it is necessary further research to analyze the applicability of those techniques over medium and large programs. If the developer has to wait too long to use a technique, it becomes useless despite of its effectiveness. Moreover, depending on the time spent to

generate the method's output, the fault could be found using the traditional debugging techniques.

3.3.5 Data-flow approaches

Chaim, Maldonado e Jino (2003) utilize data-flow testing requirements to guide for fault localization process. To achieve such a goal, they utilize the concept of *error-revealing definition-use associations* (er-dua). A tool is utilized to track the instances of duas at run-time aiming at identifying hints that might lead the developer towards the fault site. The strategy starts with the selection of suspicious duas using two heuristics. The selected duas are mapped into a piece of code and examined by the developer. If the fault is localized, the debugging process ends. On the other hand, if the fault is not in the mapped code, the developer must inspect the instances of the selected duas to find hints that lead the developer towards the fault site.

Mao et al. (2014b) and Lei et al. (2012) utilize program slicing instead of coverage data for fault localization. While spectrum-based fault localization (SFL) usually uses statement coverage data correlating with tests results, the *Slicing-based Statistical Fault Localization* (SSFL) takes into account the intersection of the *static backward slicing* and *execution slicing* of statements that affected the output of the test to identify suspicious pieces of code.

A comparison between control- and data-flow spectrum-based fault localization is studied by Santelices et al. (2009). The research focuses on the coverage of three components: statements, branches, and du-pairs (a variant of data-flow information that takes into account only c-use duas). Besides using those components individually for fault localization, the authors propose a new technique that combines the information of multiple components. It is also presented an approximate du-pair coverage, which has lower overhead than the original one which tracks du-pairs at runtime.

Alves et al. (2011) proposed an approach to reduce the *inspection cost* (number of statements that need to be inspected to find the fault) of SFL by removing some of the statements that are likely non-faulty, without increasing significantly time and memory overhead. The paper presents three techniques to achieve this goal. The first technique, called *test and dynamic slicing* (T+DS), removes the statements that are not included in the dynamic slice of the test output variable (as similarly proposed by Mao et al. (2014b),

detailed before). The second technique, called *change-impact analysis*, uses the result of the first technique to filter the statements that has been impacted by changes. The *statements impacted by changes* are the statements that affect the definition of variables that: 1) have been dynamically influenced by changes (backwards) and 2) influence the test output variable (forward). Finally, the third technique, *test and change impact* (T+CI), also uses the result from the first technique and then filter the statements that has been changed.

Wen et al. (2011) propose the *program slicing spectrum-based software fault localization* (PSS-SFL) technique, which combines dynamic slicing information and spectrum-based fault localization. The dynamic slicing part of the technique aims to reduce the number of elements by considering only elements that were executed by at least one failing test case. The authors also propose a new way to calculate the coverage matrix. Differing from the traditional technique, which only consider whether the element was executed by a particular test case, the novel approach registers the frequency that a element was executed in each test case, enabling the heuristic to be calculated differently.

Ju et al. (2014b) propose a fault localization technique based on full slices and execution slices, called *Hybrid Slice Spectrum* (HSS). The idea of this approach is to include only program entities for which the output is dynamically dependent, in other words, to exclude program entities whose execution does not interfere with the test output. For this matter, it is used a combination of full and execution slices. Furthermore, the paper also presents a new formula to calculate suspiciousness value.

Masri (2010) presented a study on fault localization based on *Dynamic Information Flow Analysis* (DIFA). DIFA comprises information flow (including variables and commands) over complex interactions between program elements. The DIFA algorithms utilizes directly dynamic control dependence (DDynCD), which includes statements that influence the execution of the target variable and dynamic direct data dependence (DDynDD), which includes variables that influence the value of the target variable. Besides these two dependences, DIFA includes the use of a returned value; the use of a value passed as a parameter; and the control dependence on an invocation instruction of a calling method. The combination of these five types of information are called *DInfluence*.

Zhang et al. (2014) utilize only the dynamic slicing of the incorrect output of a failed test case and then calculates the suspiciousness value of these statements using the traditional SFL technique. Many approaches focus on the backward dynamic slicing of

the test output. This research filters all the statements that do not affect the output of a failed test.

[Liu et al. \(2013\)](#) establish a Bayesian model using the test result and the program trace slicing. Then, they use the Bayesian Theorem to calculate the suspiciousness value of each statement. It is calculated from the probability of failing the program execution when the statement is covered.

[Ma et al. \(2013\)](#) propose a novel combined dependence network (CDN) based fault localization method. The work calculates the combined dependence probability of each node. It consists of the conditional probability (the probability of a statement to be in a certain state) and the path probability (the probability of a statement to be executed) of each node in the CDN. These two probabilities are utilized to assign a suspiciousness value.

[Cao et al. \(2014\)](#) present a fault localization technique based on dynamic slicing and association analysis. The dynamic slicing is utilized to narrow the range of the statements, then association analysis is used to calculate the suspiciousness value of each statement. Association analysis finds the correlations between the statements in the execution traces and the failed test results.

[He et al. \(2014\)](#) merge different execution paths of a program based on analysis of control-flow. The goal is to apply a reverse data dependence model so that the data dependency chain is then ranked from the most to the least suspicious of containing the fault.

[Zhang, Kim e Khurshid \(2013\)](#) present a tool, called FaultTracer, which uses program changes and extended call graphs (ECG). The ECG is a method call graph with field access information. The tool first computes the dependences between the atomic changes, then select a subset of tests which could be affected by those atomic changes based on the ECG information. Finally, the tool ranks the atomic changes using a SFL technique.

[Yang, Wu e Liu \(2012\)](#) propose a technique in which the variable trace is recorded and also combined with data dependency between those variables. This information is represented in a graph, which will be mined to identify subgraphs that are more suspicious of containing faults.

[Hofer e Wotawa \(2012\)](#) use the traditional SFL as a first step and afterwards uses probabilities of single statements using slicing-hitting-set-computation (SHSC). This

technique combines variables slices of failing test cases and minimal diagnoses to compute the fault probabilities of statements.

[Sun, Li e Ni \(2008\)](#) use execution and dynamic backward slicing of the output to filter statements and to delete similar passing tests. A dichotomy approach is presented, in which a developer has to determine if the most suspicious code has the fault or not. If the fault is not found, the developer must detect whether the values are already incorrect at these point or not. The next iteration of this approach will consider only code executed previously or posteriorly to this point based on the decision cited above.

[Yu et al. \(2011\)](#) try to minimize the overhead by using a semi-dynamic approach. It uses a static control-flow graph and a dynamic dependence graph. The backward slicing is used to analyze the dependency relationships between execution statements and execution results.

[Assi e Masri \(2011\)](#) try to identify short dependence chains that are highly correlated with failures. Differing from other studies which consider the element as a *dua*, statement, or branch, this paper also consider dependence chains (e.g. $DUA \Rightarrow BRANCH \Rightarrow DUA$).

[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 register also parameters and method-return values. To reduce the number of possible parameter values, they discretize numerical parameter and return values using Data Mining techniques.

[Wang e Roychoudhury \(2007\)](#) adopt a step-wise approach, in which a hierarchically dynamic slicing is applied at various levels of granularity. The program execution trace is divided into phases, with data/control dependencies inside each phase being not showed, only the inter-phase dependencies are presented to the developer. The developer might step to next phase if the fault is not found.

[Sun et al. \(2007\)](#) utilize dices from execution slices of a failing test case and execution slices of three passing test cases to prioritize the code to be inspected. The code then can be refined and augmented using elements of the execution slice from those test cases. These steps require the developer's intervention to stop when a fault is found.

[Wong e Qi \(2006\)](#) and [Wong e Qi \(2004\)](#) propose a program dicing (subtraction of two slices) method. Two approaches are presented, the first one tries to include additional code for inspection based on inter-block data dependency, the second approach tries to exclude less suspicious code using information of successful tests.

Agrawal et al. (1995) also present a dicing technique in which a developer uses failing and passing test cases to eliminate pieces of code that might not contain a defect and aggregate pieces of code that might contain a defect.

Summarizing, almost half of the studies used a *slicing* technique, considering that *dicing* is also a slicing technique, 61% of the papers introduced an approach based on slicing. Although it is definitely the most popular type of data-flow technique, the time and memory overhead to calculate the slices can be large because it can comprise an extensive amount of code. *Dua* is utilized by five studies. It can track less information, but it does not burden the tool with excessive data. The remaining three data-flow types (*method call with data-flow*, *block-data-dependency* and *data-chain*) augment control-flow with data-flow information. Those approaches can add more “context” information, since they mix control- and data-flow. As cited before for slicing techniques, it can carry excessive information and degrade the tool’s performance.

During the analysis of the papers it was noticed that some studies use a step-wise approach (CHAIM; MALDONADO; JINO, 2003), (SUN; LI; NI, 2008), (SUN et al., 2007), (AGRAWAL et al., 1995), in which the user has to make a decision after the tool has processed the test cases. Usually, the user has to investigate a small piece of code and tell if it contains the fault. When the fault is not found the tool take the user’s answer to make further processing.

Three studies presented data mining techniques to recognize patterns in failing test cases (LIU et al., 2013), (YANG; WU; LIU, 2012), (EICHINGER et al., 2010). Despite not being the main field of these studies, artificial intelligence occasionally has been used to address automated debugging issues.

The combination of data-flow spectrum-based fault localization with recently changed code was studied by two authors (ALVES et al., 2011), (ZHANG; KIM; KHURSHID, 2013). These approaches investigate the code that was changed from the previous version of the program to the current one. The rationale is that part of these changes might be the root of the problem, since the test cases were passing on the previous version and are not passing anymore. The downside is that the changes may trigger a failure from a infection that was caused by a not modified piece of code. The comparison between versions also requires a version control environment that fits the needs of the tool, while traditional SFL demand only the test oracle.

3.4 Conclusion

This systematic review presents an overview of fault localization techniques based on data-flow information. We selected 26 papers which used data-flow for this purpose. Despite the increasing number of studies which have been proposed for fault localization, a few of them have used data-flow information.

From our review, we observed that the use of data-flow in debugging is in its infancy. There are few initiatives that use definition-use associations (duas), while others have used program slicing, program dicing, method-call graphs, block-data-dependency, and data-chain. Step-wise approaches, data mining techniques, and change-impact analysis are also used by some authors to cope with data-flow information.

Data-flow-based techniques have presented promising results to pinpoint faults. However, in almost all works discussed, the techniques are assessed with small to medium-sized programs. Unfortunately, such programs are hardly similar to those used at industrial settings. This limitation occurs due to the high costs of collecting data-flow information. Moreover, most of the studies do not assess the time and memory overhead of their proposed techniques.

The use of instrumentation strategies with reduced overhead encourages the use of data-flow approaches in SFL techniques. The amount of information collected by data-flow approaches is larger than that of control-flow techniques. As a result, SFL techniques based on data-flow can be utilized to narrow down the most significant data-flow relationship for fault localization. Future research should tackle these issues aiming at helping to evolve the SFL area. Furthermore, the efficiency and the effectiveness of data-flow coverage applied in SFL should be evaluated by experimenting with industry-level programs.

3.5 Final remarks

This chapter described the details of a systematic review conducted on the use of data-flow information in SFL. The next chapter will present the characteristics of a tool that implements SFL techniques supported by control- and data-flow coverage.

4 Jaguar

In this chapter, we present a new tool that uses control- and data-flow coverage information for Spectrum-based fault localization (SFL). The tool ranks elements of the code (e.g., lines and definition-use associations) from the most to the least suspicious of containing the fault. Details on the implementation of the tool, as well as on the features provided to help the developer to localize the fault more efficiently, are discussed.

4.1 Jaguar architecture

We developed a new tool called Jaguar, which stands for *JAva coveraGe faUlt locAlization Ranking*. It utilizes features from different tools to perform SFL. For a better understanding, this section will be divided into three parts. The first part will detail the components in charge of invoking test cases and collecting code coverage information. The second part describes the components responsible for storing coverage data and test results, as well as for calculating the suspiciousness value of each code element. The third part will describe the software components that organize and present this information to the end user.

Jaguar overall architecture is illustrated in Figure 10. It covers all components and steps necessary to accomplish SFL with control- and data-flow information. The following sections will discuss the flow of information in Jaguar by referring to the components and steps described on Figure 10.

4.1.1 Invoking test cases and collecting coverage

As summarized above, Jaguar invokes unit tests of the subject program and collects the code coverage information for each element (node, branch or dua). These tasks are discussed together because they are related to the main purpose of collecting statistical data needed to perform SFL.

SFL techniques rely on test cases. Therefore, one needs to localize and run all tests cases of the faulty program. The test cases must be JUnit tests, either unit tests or integration tests, but only the coverage of the local project will be collected. The *Jaguar Eclipse Plug-in* contains the *Java Launch Configuration Delegate* which makes the *JUnit*

Figure 10 – Jaguar architecture

Source: Henrique Ribeiro, 2016

Runner features automatically supported. The user can select the test folder of any Java Project imported on Eclipse that contains JUnit Tests to run using the Jaguar Plug-in. A configuration tab allows the user to select the type of code coverage (Data-flow or Control-flow) to be collected from the test runs.

Prior to test case execution, it is necessary to set the configurations that will guarantee the code coverage information to be collected. The Java programming language provides services that allow programs running on the Java Virtual Machine (JVM) to be instrumented by another program. Program instrumentation consists of modifying the original code by inserting additional code to collect coverage information during its execution. Hence, a *Java Agent* must be set on the *JVM Arguments* to instrument the classes used by the unit tests and then generate the coverage information.

JaCoCo¹ is an open-source coverage tool for Java. It is able to determine the coverage of instructions, branches, lines, methods and classes of a program under test. Therefore, after a test case or a test suite executed, JaCoCo can report which of these elements were executed for each class. JaCoCo tracks only control-flow information of the code (lines and branches); data-flow information is not available in this tool.

BA-DUA (ARAUJO; CHAIM, 2014) is a recent code coverage tool for Java which also makes use of Java Agent services to instrument the program and collect coverage

¹ <http://www.eclemma.org/jacoco/>

information. Differently from JaCoCo, BA-DUA focuses only on data-flow information. It provides data coverage of intra-procedural definition-use associations (dua) of each variable executed by a program.

The tool presented in this proposal, Jaguar, utilizes a modified version of JaCoCo, called JaCoCoPlusBadua², which includes the BA-DUA library. It uses the communication structure of JaCoCo to exchange information with BA-DUA. JaCoCoPlusBadua allows to specify whether the coverage information should be data-flow (BA-DUA) or control-flow (JaCoCo).

As symbolized by Step 1 (the number 1 on Figure 10) the *JaguarRunner* will invoke *JaguarCore* passing the parameters needed and also invoking JaCoCoPlusBaDua as the Java Agent. The parameters include (1) the path of the file containing the list of test that have to be executed, (2) the project path, (3) the source code path, and (4) the type of coverage.

From this point on, all unit tests will be executed sequentially (Step 2). Jaguar implements a *JUnitRunListener* which will be called every time a test case is started and finished, passing information about the test case execution (e.g., the outcome of the test—pass or fail). At the end of each unit test, *JaguarCore* will send a command to *JaCoCoPlusBaDua* using a local TCP connection (Step 3) asking for the coverage information and requesting to reset all the coverage data. As a Java Agent, *JaCoCoPlusBaDua* instruments and collects coverage information while the unit test is executed. Hence, it contains the coverage information of each element. The coverage data is then sent to *JaguarCore* through the TCP connection (Step 4). We detail how this information is received and how it is used further on.

4.1.2 Storing and calculating

SFL techniques requires that the four coefficients are determined for each code element to calculate its suspiciousness. The number of *failed* test cases that *executed* element j ($c_{11}(j)$), the number of *passed* test cases that *executed* element j ($c_{10}(j)$), the number of *failed* test cases that did *not execute* element j ($c_{01}(j)$) and the number of *passed* test cases that did *not execute* element j ($c_{00}(j)$).

² <https://github.com/henriquelemos0/jacoco>

At this point, Jaguar receives an object with all the coverage data regarding the test case. Jaguar analyzes this object to iterate over all the classes, methods and then lines or duas. A list of elements (lines or duas) are managed to store the coverage of each element for all the test cases. Each element is updated to register whether it was executed in a failing test case or in a passing test case (Step 5 in Figure 10).

The Steps 2, 3, 4, and 5, respectively, *Execute Unit Test*, *Ask for coverage information*, *Receive coverage information* and *Add coverage data* are executed sequentially N times, N being the total number of test cases.

After all the test cases were executed, Jaguar calculates the coefficients and the suspiciousness value of each element (Step 6). Jaguar does not keep the four coefficients of each element during the collection of the coverage information. Only after all the tests have been executed Jaguar calculates two of them. Two global variables regarding all the test cases register the number of tests that have been executed (**nTests**) and the number of tests that have failed (**nTestsFailed**). Each element keeps two variables to register (1) the number of times it was executed in a failing test case (**cef**) and (2) the number of times it was executed in a passing test case (**cep**). The other two coefficients that represent (1) the number of times it was not executed in a failing test case (**cnf**) and (2) the number of times it was not executed in a passing test case (**cnp**) are calculated when all the tests have been executed using the following equations: $\text{cnf} = \text{nTestsFailed} - \text{cef}$ and $\text{cnp} = \text{nTests} - \text{nTestsFailed} - \text{cep}$.

With these coefficients, Jaguar is able to calculate the suspicious value of each element applying one of the known heuristics. Currently, all the ten heuristics (*DRT*, *Jaccard*, *Kulczynski2*, *McCon*, *Minus*, *Ochiai*, *Op*, *Tarantula*, *Wong3*, *Zoltar*) are already implemented. The cost of determining the suspiciousness values is very low in comparison to the time for test suite execution and coverage data collection. The suspicious value of each element is calculated by iterating the list containing all the elements covered by the test cases, and then applying the specified heuristic.

Henceforth, Jaguar holds all information needed to apply a SFL technique. This data can be used in different ways to validate and measure the effectiveness and efficiency of a SFL technique using control- or data-flow coverage information.

4.1.3 Results

The final task of Jaguar is to present the suspiciousness of the code elements (lines and duas) in different ways to facilitate the use of it in fault localization.

Jaguar saves the objects containing the elements information and suspicious values in an XML (EXtensible Markup Language) file, Step 7 in Figure 10. This approach allows other programs to use the data since it can be loaded to an object in any language to further processing or reporting.

After all the steps described above are executed, the user can run the *Jaguar View*. This action will trigger *Jaguar Plug-in* to read the XML file that contains the coverage data; this task is represented by Step 8 on Figure 10. Jaguar View reads each coverage element, Step 8, and then present that information in a way that the developer can browse the code and see which element are the most suspicious, Step 9 of Figure 10.

When the user run Jaguar (Step 1 described early in this chapter) it also have to chose how the resulted coverage information should be structured, *Flat* or *Hierarchical*. The Flat option means that the elements will be ordered regardless of which package and class it comes from. The Hierarchical option will result in a outcome that ranks the packages, class, methods and then its elements.

Those two outcome options make it possible to view the results of the SFL in two ways. One is called *Roadmap* and the other is refereed as *Hierarchical*. The former will make use of the Flat outcome and the later will use the Hierarchical one. They are better detailed below.

4.1.3.1 Roadmap

Jaguar View reads each coverage element and order it by methods, showing a window that contain all the methods ordered from the most to the least suspicious. It can be seen on the right top corner of Figure 11. A window below it will show the duas or lines (depending on the choice made when running the test suite) also ordered from the most to the least suspicious. The contents of this last window will change based on the method selected on the top window. In that way the user can see the most suspicious elements of each method. When a element (dua or line) is selected, the window that shows the source code (in the center of Figure 11) will open the Class and focus on the line that contain

the element. Besides the automatic source code focus, the *Jaguar View* will change the background color of each line based on its suspiciousness. The most suspicious lines will have a red background; the medium suspicious lines will have a yellow; the lines that were covered but are less suspicious will be in a green background; and finally the lines that have no elements covered will be in a gray background.

Figure 11 – Jaguar View - Flat

Source: Henrique Ribeiro, 2016

4.1.3.2 Hierarchical

The Hierarchical option is equal the Roadmap one in terms of coloring and source code selection, but it differs on the way the elements are presented in the window on the top right corner. In this option, all the packages are presented sorted by suspiciousness. When the developer selects a package, its classes will be presented underneath it. Likewise, the methods of a selected class are listed. All the elements of these levels (package, class and methods) are sorted by the most to least suspicious element.

When a method is selected, the duas or lines are presented on the window below it. A preview of the Hierarchical view can be seen on Figure 12.

Figure 12 – Jaguar View - Hierarchical

Source: Henrique Ribeiro, 2016

4.2 Final remarks

This chapter presented the details regarding the architecture of the Jaguar tool implemented in this work. The current version of Jaguar runs unit tests and generates an XML file with the suspiciousness assigned to lines and duas according to ten different heuristics. It then shows the resulted coverage information in a graphical interface. With the graphical interface, the developer is able to browse the code and the suspicious elements. The next chapter will present the experiments executed to assess the control- and data-flow coverage efficiency and effectiveness.

5 Experimental Assessment

This chapter details the experimental assessment conducted to evaluate the research questions established in this work. We start off presenting the experimental design and the results of the experiments. We finish up with a discussion.

5.1 Experiment design

We use the concept of *effort budget* to assess the effectiveness of the techniques. An effort budget is given by the absolute number of lines a developer investigates before abandoning a technique. We utilized different effort budgets for the experiments, which varied from 5 to 100 lines.

The *rationale* is that if the developer is unable to find the bug by investigating the number of lines established in a particular effort budget (e.g., 20 lines), then the technique offers little help in locating the fault. The effectiveness was assessed by the number of bugs located within each effort budget, independently of the total size of the program.

5.1.1 Research questions

Which heuristic is more effective to support an SFL technique based on control-flow coverage?

The 10 heuristics presented in Section 2.3 are compared against each other for seven effort budgets (5, 10, 20, 30, 40, 50, and 100). For each budget, we verify which heuristic reached more bugs. The goal is to identify whether there is a heuristic that performs better with control-flow coverage.

Which heuristic is more effective to support SFL technique based on data-flow coverage?

This question is analogue to the previous one; however, it regards the data-flow coverage. The 10 heuristics are also compared using the seven distinct effort budgets. The goal is to identify whether there is a heuristic that performs better with data-flow coverage.

What coverage type locates more bugs: control- or data-flow coverage?

The heuristics are sorted by the amount of bugs located, considering the two coverages and the seven different budgets. A defect is considered as found if the total number of blocks needed to find the fault is smaller than the maximum budget established for that experiment. The results indicate which heuristic performs better for the subjects of the experiment.

What coverage type ranks the bugs better: control-flow or data-flow coverage?

For each heuristic and budget, the number lines of code to reach the bug using control- and data-flow coverages is compared. The goal is to assess which coverage ranks better the buggy lines

What is the costs associated with the use of control- and data-flow coverages in SFL?

The time spent to generate the elements sorted by suspiciousness is assessed. It comprehends the tasks of running all test cases, storing the code coverage for each test case and then calculating the suspicious value of each element using one of the heuristics. The results of each coverage are compared to the time spent running only the test suite, without any coverage, as a baseline.

5.1.2 Procedure

Selected programs

For the experimental assessment, it was used six different programs, in which four of them (JFreeChart, Commons Lang, Commons Math, Joda-Time) were extracted from the Defects4J Database (JUST; JALALI; ERNST, 2014). Table 8 lists all programs and the correspondent number of lines of code and number of test cases. The column *KLoc* represents the most recent version size in thousands of lines of code, as reported by SLOCCount¹. The column *Test Cases* represents the most recent version of the test suite size. The column *Real ?* indicates whether the bugs are real; that is, collected during the

¹ <https://sourceforge.net/projects/sloccount/>

development of the program, or seeded for experimental purposes. The only program with seeded faults is Ant; it was obtained from the SIR repository².

Table 8 – Programs characteristics

Program	KLoc	Test Cases	Real?
Ant	79	986	No
JFreeChart	96	2,205	Yes
JSoup	10	468	Yes
Commons Lang	22	2,245	Yes
Commons Math	85	3,602	Yes
Joda-Time	28	4,130	Yes

Source: Henrique Ribeiro, 2016

The characteristics of the programs and the test suite may influence the efficiency and effectiveness of the SFL technique. More test cases may help improve the suspiciousness accuracy since the suspicious value assigned to an element will represent better its influence in passing and failing test cases; and bigger programs tend to slow down the code coverage task, as more data need to be collected.

Selection of defects

For each program, many versions with seeded and real defects were selected to be tested. Table 9 summarizes the total number of versions for each program. The table is divided into three groups referred to in columns *Version*, *All multiple lines*, and *Data-flow Limited multiple lines*:

Version. This group represents the faulty versions as they are available in the repositories.

A single defect, though, may be spread in several lines; that is, it was needed changes in more than one line to fix the bug. In total, there are 165 different bugs in the selected programs.

All multiple lines. This group deems each line of a multiple-line defect as a different bug. The rationale for this set is to capture situations in which the developer misses a well positioned buggy, but still can locate the defect in the other buggy lines. For

² <http://sir.unl.edu>

that matter, the pair heuristic and coverage should position well the other buggy lines.

Data-flow Limitation multiple lines. This group encompasses the same buggy versions of the *All multiple lines* group, excepting those versions for which data-flow coverage was not complete due to BA-DUA limitations. Latter in this section, we discuss the limitations of the BA-DUA tool in collecting data-flow coverage.

Table 9 – Program versions

Program	Version	All multiple lines	Data-flow limited multiple lines
Ant	14	15	10
JFreeChart	26	45	26
JSoup	38	42	38
Commons Lang	20	37	30
Commons Math	40	66	43
Joda-Time	27	60	26
Total	165	265	173

Source: Henrique Ribeiro, 2016

We tried to keep the number of defects per program relatively equivalent among the programs. For JFreeChart and Joda-Time, it was used all the versions available in the Defects4J Database; for Commons-Math it was randomly selected 40 out of the 106 available versions; Commons-Lang had 67 versions, but only the first 20 were used due to problems found to build the projects, mainly because of compatibility problems with old Java versions (e.g., 1.3); for JSoup and Ant were used versions already prepared for other fault localization experiments by our research group (SAEG).

The *All multiple lines* group (third column of Table 9) represents each buggy line of a faulty version as a different and distinct bug. Let us suppose that a bug in Ant version 1.0 required to change two different classes in two different lines (e.g., Class1 Line 50 and Class2 Line 20) to completely fix the bug. In *All multiple lines* group, there will be two different faulty versions of Ant: one in which the fault is located at Line 50 of Class1 and another at Line 20 of Class 2. The same code and coverage data are used, but two different defects are taken into account. This strategy was used to assess the rank position of each buggy line. In the first case, the developer needs to reach the first class to locate the bug; in the latter case, s/he needs to reach the second class to locate it. In doing so, we are

able to assess whether the pair heuristic and coverage is effective in reaching any of the buggy lines.

The group called *Data-flow limitations multiple lines* (fourth column of Table 9) contains all the versions from the previous group excluding those for which BA-DUA is unable to collect reliable data-flow coverage. There are two cases: 1) there is a non-handled exception; and 2) the bug is in a single-block method. The first case happens because BA-DUA marks the exercised duas whenever the method is exited; in this case, the method is exited in a non-predictable way so that BA-DUA library does not mark the exercised duas as covered. The second case occurs because there is no dua when the definition and use occur in the same basic block; hence, there is no dua in the buggy method. These two situations are due to the lightweight manner that BA-DUA handles coverage information. Indeed, the former case can be easily handled if BA-DUA adds a dua in single-node methods, but it will increase BA-DUA's overhead. Hence, we created this group which excludes these two cases for a fair comparison between the control-flow and data-flow results.

5.1.2.1 Data collection

Jaguar was used for the data collection of this experimental assessment. Using the Jaguar Eclipse Plug-in interface, we run the JUnit tests of the selected project, collected the coverage data, generated the proper matrix of coefficients and finally obtained the suspicious value of each element (line or dua). This procedure was executed for each faulty version to collect control- and data-flow coverages.

5.1.2.2 Bug localization

To assess whether the bug was localized, first we identified where it is located. The Defects4J Database programs use GIT³ as the version control system; therefore, we were able to compare the differences between the buggy commit and the fix commit. With that information in hand, we verified the lines changed to fix the bug. JSoup was extracted

³ <https://git-scm.com/>

from the project public GIT repository. Ant was the only project with seeded defects; it was extracted from the SIR repository ⁴.

When code was added to fix a bug, the previous line of code (not including comments and blank lines) was deemed the bug site. However, if the previous line was a code with no commands (for example, closing brackets of a *if* or *for*, or the signature of the method), the line after the change was treated as the faulty line.

Once the buggy class and line of each version is determined, its position in the rank of suspicious elements should be checked. A script was written to search over each coverage output file (generated by Jaguar) to determine if the buggy line was among the first N lines, being N the maximum number of lines of a budget. The search is successful if there is a match in less than or equal to N lines; otherwise, it is unsuccessful. In addition, the number of lines needed to inspect up to finding the bug is recorded.

The data-flow elements are duas; thus, they should be mapped onto lines to get the final lines to search for the fault. A dua always has a *definition* line and a *use* line and possibly a *source* line (if it is a p-use dua). To check whether the bug was found, the faulty line is compared to those three lines (definition, use and source). When this check resulted as false, all the three lines (or two, when there was no source) are added to the number of lines needed to be inspected until finding the fault.

When ties occur, the worst case is considered. So, if two or more lines have the same suspicious value, the number of lines needed to be inspected to find the fault includes all of them. This is so because there is no guarantee that the buggy line will be the first or the last to be inspected.

5.1.2.3 Budgets

The effort a developer allocates to a 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. To assess how the coverages and the heuristics perform under these different scenarios, seven budgets were chosen: 5, 10, 20, 30, 40, 50, and 100 lines.

⁴ <http://sir.unl.edu>

5.1.3 Statistical Analysis

The vector containing the results of each pair (heuristic, coverage) for each budget was tested to check whether the data follow a normal distribution. This test for all the vectors returned false. Thus, to evaluate the significance of effectiveness between the two coverages and among the heuristics, we applied the paired Wilcoxon-Signed-Rank, which is a non-parametric statistical hypothesis test for data that do not follow a normal distribution. A significance level of 5%, hence a $p - value$ smaller than 0.05, were expected. To carry out these tests, a script using the language R ⁵ was developed containing twenty vectors for each budget, one for each pair (heuristic, coverage), and then the call to the Wilcoxon test.

5.2 Results

The presentation of the results is two-fold. First, the effectiveness is presented using barplots. The number of bugs located within each budget for different pairs of heuristics and coverages is plotted. Next, we describe the statistical tests comparing control- and data-flow coverages and comparing heuristic against heuristic for each coverage type. The statistical tests assess whether one technique needs to inspect more lines of code than the other in a particular budget. A tie occurs whenever both techniques inspect a number of lines greater than the budget.

5.2.1 Control- and data-flow effectiveness: barplots

Figure 13 shows the data for the control-flow coverage. Each budget is grouped and the heuristics are sorted as the legend present them. The figure shows that Ochiai is better when the maximum budget is used. For all the other budgets, Kulczynski2 and Mccon are the best heuristics to locate bugs, among the 10 heuristics studied in this work.

Figure 14 presents the data for the data-flow coverage. A similar behavior is present in both control- and data-flow effectiveness data. Kulczynski2 and Mccon are the best heuristics to locate bugs for small budgets. Ochiai is slightly better using data-flow coverage from budget 30 onwards.

⁵ <https://www.r-project.org/>

Figure 13 – Effectiveness of heuristics using various budgets for control-flow.

Source: Henrique Ribeiro, 2016

Both figures present data obtained from the *Data-flow limitation all multiple lines* group since the idea is to compare the two coverages.

5.2.2 Control- and data-flow: statistical tests

Statistical tests compare the control- and data-flow effectiveness in terms of number of lines to reach a fault. We applied the paired Wilcoxon test for all the 10 heuristics and the seven budgets, using the group *Data-flow limitations multiple lines*. We utilized this group to allow a fair comparison between coverages.

The null hypothesis is that there is no difference between control- and data-flow coverages. The alternative hypothesis is that control-flow coverage requires the examination of more statements than data-flow coverage. The p-values (already converted to percentage)

Figure 14 – Effectiveness of heuristics using various budgets for data-flow.

Source: Henrique Ribeiro, 2016

of all tests are summarized in Table 10. The value **2.023** in the first row and fourth column shows that control-flow needs more lines to be inspected to locate faults than data-flow with confidence level of **97.977%**. The significant p-values are printed in boldface.

From the budget of twenty to one hundred lines, control-flow needs to inspect more lines to reach the fault for all the 10 heuristics studied with confidence level of 5%.

5.2.3 Heuristic versus Heuristic

We applied the paired Wilcoxon test among heuristics for all budgets and for each coverage (control- and data-flow). Table 11 summarizes the results for control-flow and Table 12 for data-flow. The statistical tests of the control-flow coverage utilized the *All multiple lines* group because the goal is to compare the ability of the heuristics in locating

Table 10 – Control-flow versus data-flow effectiveness

Heuristic	5	10	20	30	40	50	100
DRT	75.710	20.470	2.023	0.337	0.482	1.368	1.955
Jaccard	84.150	38.970	1.798	0.387	0.120	0.268	0.411
Kulczynski2	87.390	37.670	2.323	0.969	0.822	1.455	0.741
Mccon	87.390	37.670	2.323	0.969	0.822	1.455	0.741
Minus	68.790	11.030	1.136	0.323	0.390	1.039	1.711
Ochiai	83.640	36.870	0.697	0.213	0.125	0.309	0.649
Op	75.710	23.670	2.903	0.544	0.530	1.340	1.773
Tarantula	76.340	36.570	0.611	0.181	0.067	0.168	0.142
Wong3	76.880	2.712	0.179	0.305	0.227	0.148	0.041
Zoltar	88.100	38.250	2.629	1.287	1.159	2.059	0.823

Source: Henrique Ribeiro, 2016

bugs with control-flow coverage. The tests with data-flow coverage used the *Data-flow limitation all multiple lines* group because it contains reliable data. Appendix B presents the p-values of all tests carried out.

Table 11 and Table 12 describe the paired Wilcoxon test of the *row* heuristic against the *column* heuristic. The null hypothesis is that there is no difference between the *row* heuristic and the *column* heuristic. The alternative hypothesis is that the *row* heuristic requires the examination of more statements than the *column* heuristic. In other words, the *column* heuristic performs better than the *row* heuristic. The contents of the cells are those budgets for which the paired Wilcoxon test rejected the null hypothesis with 5% of significance or less. Whenever the cell is empty, the null hypothesis could not be rejected. The contents is “—” when a heuristic is compared to itself, which does not make sense.

For instance, the contents of row *DRT* and column *Jaccard* of Table 11 is *30-100*. It means that DRT inspects more lines than Jaccard to hit the bug using control-flow coverage for budgets 30, 40, 50, and 100 with statistical significance; that is, Jaccard performs better than DRT for these budgets with control-flow. The content of row *Jaccard* and column *Kulczynski2* is *5,10* meaning that Kulczynski2 performs better than Jaccard using control-flow coverage with statistical significance for budget 5 and 100.

To identify the *winning* heuristics, one should look at those columns with more non-empty cells. On the other, the *losing* heuristics are identified by the rows with more non-empty cells. For control-flow (Table 11), Wong3 is a losing heuristic since it is unable to perform better against any other heuristics. On the other hand, Ochiai is a winning heuristic but for budgets above 30 lines; Jaccard has a behavior similar to Ochiai’s.

Kulczynski2 and Mccon, in turn, perform well against other heuristics for small budgets (5 and 10).

Table 11 – Heuristic versus heuristic: results for control-flow

Heuristic	DRT	Jaccard	Kulcz.	Mccon	Minus	Ochiai	Op	Taran.	Wong3	Zoltar
DRT	—	30-100				30-100				
Jaccard		—	5,10	5,10						
Kulcz.			—			100				
Mccon				—		100				
Minus		20-100	5-50	5-100	—	30-50				5,10
Ochiai						—				
Op		30-100				30-100	—	100		
Taran.		20-50	5,10	5,10		10,100	100	—		5,10
Wong3	5-100	5-100	5-100	5-100	5-100	5-100	5-100	10-100	—	5-100
Zoltar		100				100				—

Source: Henrique Ribeiro, 2016

Table 12 contains the data comparing heuristic against heuristic using data-flow coverage for each budget. For data-flow, Wong3 is a losing heuristic because it does not perform better than any other; Tarantula has a similar behavior performing better only against Wong3. Kulczynski2 and Mccon, again, perform well against other heuristics for smaller budgets (20 lines). Ochiai is the most winning heuristic, but its performance improves from 30 lines onwards.

Table 12 – Heuristic versus heuristic: results for data-flow

Heuristic	DRT	Jaccard	Kulcz.	Mccon	Minus	Ochiai	Op	Taran.	Wong3	Zoltar
DRT	—	100	20-100	20-100		30-100				40-100
Jaccard		—				100				
Kulcz.			—							
Mccon				—						
Minus		100	20-100	20-100	—	30-100				100
Ochiai						—				
Op		50-100	20-100	20-100	50-100	30-100	—			40-100
Taran.	5-10	10-100	5-50	5-50	5-10	5-100	5-10	—		5-20
Wong3	5-100	10-100	10-100	10-100	5-100	10-100	5-100	20-100	—	10-100
Zoltar										—

Source: Henrique Ribeiro, 2016

5.2.4 Efficiency

Besides the effectiveness of the coverages and heuristics, we measured the time spent determining the suspiciousness values using control- and data-flow coverages. This task comprehends running each unit test, storing the coverage information and finally calculating the suspiciousness of each element. Table 13 presents in the first column the name of the project; from the second to fourth columns, the time (in seconds) spent executing only the JUnit test (to be used as baseline), the time to collect the control-flow suspiciousness and the time to collect the data-flow suspiciousness, respectively, are presented. These values consist of the average execution time for all versions of a project belonging to the *All Multiple Lines* group (9) to collect all 10 heuristics utilized in the assessment.

The column *CF Over.* presents the overhead to collect data-flow coverage compared to the baseline (only JUnit). The column *DF Over.* shows the overhead to collect data-flow information also compared to the base line (only JUnit). The last column, named *DF/CF*, shows the ratio between the data-flow time and data-flow time, to indicate how much extra time data-flow needs compared to control-flow.

As an example, the first project, Ant, takes about 71 seconds to have only the JUnit test executed. The same project takes about 78 seconds to have the JUnit test executed and the control-flow data collected, which is 12.02% more costly than only the JUnit tests. It then takes about 97 seconds to have the JUnit test executed while the data-flow coverage information is collected, which is 35.82% more costly than only the JUnit tests and 23.54% more costly than the JUnit and control-flow information took.

Table 13 – Control-flow and Data-flow efficiency for each project

Project	JUnit (s)	CF (s)	DF (s)	CF Over.	DF Over.	DF/CF
Ant	71.570	78.659	97.178	12.02%	35.82%	23.54%
JFreeChart	22.267	36.363	88.623	70.11%	309.94%	143.72%
JSoup	4.350	11.370	22.947	197.29%	490.83%	101.82%
Commons Lang	18.835	33.920	89.670	131.81%	538.71%	164.35%
Commons Math	144.318	265.165	515.950	89.22%	301.80%	94.58%
Joda-Time	4.883	47.935	165.422	881.95%	3298.06%	245.09%

Source: Henrique Ribeiro, 2016

As can be seen, for some project, the data-flow coverage costs only 23.54% more than control-flow coverage. For other projects, like Joda-Time, data-flow may cost up to 245% more than control-flow.

5.3 Discussion

This section discusses the results presented in the tables and figures from the previous section. The discussion is organized by and aims to address the research questions.

Which heuristic is more effective to support an SFL technique based on control-flow coverage?

There was no heuristic that presents better results than all the other heuristics for all budgets as shown in Table 11. However, Wong3 had the worst performance, with confidence level of 5%, in comparison with all the other nine heuristics for all budgets, excepting for budget 5 with Tarantula. Considering only low budgets (5 and 10), Kulczynski2 and McCon present the best results, outperforming four other heuristics (Jaccard, Tarantula, Minus and Wong3) with statistical significance. Looking to mid range budgets (30, 40 and 50), Jaccard and Ochiai surpass five competitors (DRT, Minus, Op, Tarantula and Wong3). For the last analyzed budget (100) Ochiai is certainly the best choice, it presents better results than all the others (except for Jaccard) with statistical confidence.

In terms of total number of faults located, as shows in Figure 13, Kulczynski2 and McCon are more effective for budgets between 5 to 50. For 50 and 100, Ochiai was more capable of locating bugs. Hence, our data suggest that for control-flow coverage Kulczynski2 and McCon are better for small budgets, while Ochiai and Jaccard are indicated for mid budgets, and Ochiai ranks better for big budgets.

Which heuristic is more effective to support SFL technique based on data-flow coverage?

The behavior of the heuristics for data-flow coverage is similar to that of control-flow coverage. There is no prevalent heuristic. Likewise control-flow, Wong3 does not perform well using data-flow coverage information. It is worst than all heuristic for all budgets, with statistical confidence, excepting for Tarantula with budget 5. Kulczynski2 and McCon

have a better performance than five heuristics (DRT, Minus, Op, Tarantula and Wong3) for budget 20; Ochiai joins them from budget 30 to 50 winning the same five heuristics. For budget 100, Ochiai surpasses six heuristics (DRT, Jaccard, Minus, Op, Tarantula and Wong3).

The total number of faults located per budget (Figure 14) shows that Kulczynski2 and McCon are slightly better than Ochiai up to budget 20. For budgets above 30, it is the opposite: Ochiai is slightly better. Thus, similarly to control-flow, Kulczynski2 and McCon seems to be better for small and mid budgets (5 to 20) with data-flow coverage, and Ochiai should be used for mid and high budgets (30 to 100).

What coverage locates more bugs: control- or data-flow coverage?

Table 14 summarizes the total number of located bugs by each coverage and budget. Column one lists the budgets utilized in the assessment; columns two, *CF*, and three, *DF*, shows the total number of bugs located within the budgets for control- and data-flow coverages, respectively. The last column is the difference between the two coverages in percentage. For budget 5, control-flow locates 26 bugs while data-flow locates 24 (which is -7.7% less than control-flow). For this first budget, data-flow performance is affected by the characteristics of the data-flow elements tracked—definition-use associations (duas). A dua consists of a definition node, a use node, and possibly a source node, which makes the developer check two or three lines just for the best ranked dua. On the other hand, the best ranked control-flow element, a line, makes her or him verify only one. If the dua locating the bug is not in the first or second position, the bug will not be located within the first five lines.

In the budget 100, data-flow does not perform better than control-flow. The reason for this result is similar to the that for budget 5. If the faulty dua is not well positioned in the rank, every previous dua will require two or three lines to be inspected by the developer, exhausting the 100 lines budget.

For budgets between 10 and 50, data-flow locates more defects than control-flow, finding from 5.8% to 22.7% more bugs. Therefore, excluding extreme budgets (like 5 and 100), our results suggest that data-flow coverage is a better choice than control-flow coverage.

Table 14 – Control-flow and Data-flow located faults

Budget	CF	DF	DF/CF
5	26	24	-7.7%
10	44	54	22.7%
20	73	83	13.7%
30	84	95	13.1%
40	96	102	6.3%
50	103	109	5.8%
100	125	117	-6.4%

Source: Henrique Ribeiro, 2016

What coverage ranks the bugs better: control-flow or data-flow coverage?

Table 10 represents the comparison between control-flow and data-flow ranks. The results show that control-flow needs a greater number of lines than data-flow to find the fault, when comparing each faulty version paired, with a confidence level of 5%, for budgets equal or greater than 20.

Despite of what was described in the last question, data-flow is significantly better than control-flow even for budget 100. That is due to the paired comparison of the Wilcoxon statistical test. When data-flow misses the fault, it is assigned the value 100. But when both control- and data-flow hit the fault, control-flow requires the inspection of more lines. As a result, data-flow needs the inspection of less lines of codes, ranking better the bug.

What is the costs associated with the use of control- and data-flow coverages in SFL?

Table 13 summarized the cost, in execution time, for each project. The data-flow requires from 23.54% to 245.09% more time than the control-flow. This result is not similar to the 38% overhead presented in the motivation of this work (ARAÚJO; CHAIM, 2014). That discrepancy is justified by the extra work needed to implement an SFL technique.

SFL techniques require that the coverage for each test case be stored before proceeding to rank calculation. The coverage needs to be collected and dumped for every unit test at run-time. Data-flow implies more information than control-flow: a dua consists

of a variable name, a definition line, a use line and possibly a source line. Moreover, a program has more *duas* than lines. All these data need to be stored by the Java Agent, and then passed to Jaguar. This extra amount of information impacts the cost of using data-flow coverage in SFL.

5.4 Threats to validity

We discuss three threats to validity: internal, external, and constructing validity. Internal validity regards the experimenter bias. The internal threats to our experiment is the Jaguar tool used to run the tests, collect the coverage data and generate the suspicious values. The implementation of this tools was manually checked using small programs. Due to the size of the programs, the data collected for the experimental assessment were not manually checked. Thus, bugs in the implementation of the tool might influence the results. The data structure used to collect and store the data-flow information causes Jaguar to consume a great quantity of memory. We chose Java collection library to implement the data structure to simplify the implementation. A more memory efficient implementation, though, might improve the data-flow efficiency results. Thus, we caution the reader that these results can be improved. The chosen strategy aimed to get a first picture of the performance benefits from the recent promising data-flow coverage tool (BA-DUA).

External validity relates to the generalization of the presented results. We used seven programs from different areas (software engineering, text and graphics processing, and mathematical functions) and sizes (10 to 96 KLOC) to expose the techniques to different contexts. Although the programs utilized in the experimental assessment are quite heterogeneous, we caution the reader that the techniques may present different results for a different set of programs.

The 173 faulty versions of the programs contains a single fault. Ant was seeded with bugs while Commons-Math, Commons-Lang, JSoup, JFreeChart and Joda-Time have real bugs found in the source code repository. The detection of the bug site for real faults were made manually based on the differences of the source code before and after the fix. We cannot guarantee that all the changed code is related to the bug fix. It was assumed that each changed code was to fix the bug (excluding changes that does not affect the program behavior such as add or remove empty lines and line indentation changes).

More experiments using programs with multiple faults should be carried to obtain more accurate results. Nevertheless, our strategy of deeming each modified line of a fix as a buggy line emulates a more realistic scenario. Another solution to deal with multiple faults is to identify test cases that detect particular faults to support program debugging (JONES; BOWRING; HARROLD, 2007). These techniques select test cases to narrow down a particular fault. Control- and data-flow in SFL can benefit from these approach.

The constructing validity concerns the suitability of the effectiveness metric. We assess the techniques' ability of finding the buggy line within a specific effort budget (5, 10, 20, 30, 40, 50 and 100 lines). Although the effort budgets were chosen arbitrarily, lower budgets replicate real scenarios for debugging techniques. One issue, though, regards the use of the list of suspicious lines. We assume that the ranked elements will be followed as they were presented, which may not actually happen in practice.

Our experiment was built to evaluate how quickly a technique will reach the fault site. Reaching the bug site does not necessarily mean locating the defect. The perfect bug detection assumption, which assumes that the developer will identify the faulty line only by inspecting it, is not guaranteed in practice (PARNIN; ORSO, 2011).

5.5 Final remarks

In this chapter, we have presented an experimental assessment of the use of control- and data-flow coverage in SFL using the Jaguar tool. We presented and discussed the results of the experimental assessment. Both control- and data-flow have similar behavior with respect to the heuristics utilized. There is no prevalent heuristic but the results suggest that Kulczynski2 and Mccon perform well for small budgets (5 to 20 lines) and Ochiai is best fitted to larger budgets (above 30). Moreover, data-flow seems to be more effective than control-flow for mid-sized ranges (20 to 40 lines), locating more bugs and requiring the inspection of less code than control-flow. However, for our implementation of control- and data-flow SFL in Jaguar, the cost of data-flow is still significantly more expensive in comparison to control-flow, varying from 23% to 245% more costly.

In the next chapter, the summary of the results achieved, our contributions, and the future work are presented and discussed.

6 Conclusions

In this chapter we present the summary of the results, our contributions and the future work.

6.1 Summary

Spectrum-based fault Localization (SFL), or Coverage-based Fault Localization, has been studied by many researchers to reduce the time and effort spent on debugging. Different code elements (e.g., statements, branches, definition-use associations — duas) are used to select the most suspicious excerpts of a program. Due to its low cost, control-flow (statement and branch) coverage has been often utilized in SFL techniques. However, data-flow (dua) coverage obtained better results in the few assessments conducted despite of its higher cost. This work compared the effectiveness and efficiency of control- and data-flow coverage in SFL. In particular, we utilized recently developed tools that reported low overhead, especially for data-flow coverage. Programs with similar size and characteristics to those developed in the industry were used in our experimental assessment.

We developed a tool — called Jaguar (*J*AVA *c*overa*G*e *f*aULT *l*ocAlization *R*anking) — that implements SFL techniques using control- and data-flow coverage. Jaguar obtains control- and data-flow coverage from JaCoCo and BA-DUA tools, respectively. JaCoCo¹ is a popular control-flow coverage tool used largely in the industry to assess the quality of test suites (MOIR, 2011). BA-DUA, in turn, collects efficiently data-flow coverage (ARAUJO; CHAIM, 2014). Jaguar was utilized on 173 faulty versions of programs with real and seeded defects. These programs were from different areas (software engineering, text and graphics processing, and mathematical functions) and varied in size from 10 to 96 KLOC. Ten known heuristics were used to rank the suspicious elements, considering seven effort budgets. An effort budget is given by the absolute number of lines a developer investigates before abandoning an SFL technique. We utilized different effort budgets for the experiments: 5, 10, 20, 30, 40, 50, and 100 lines.

Our findings suggest a similar behavior of both coverages with respect to the ten heuristics utilized to rank code elements. Three heuristics presented better performance: Kulczynski2 and Mccon had better results in small budget (5 to 30 lines); Ochiai performed

¹ <http://www.eclemma.org/jacoco/>.

better when more lines were inspected (30 to 100 lines). Regarding the control- and data-flow comparison, data-flow located more defects in the range of 10 to 50, being up to 22% more effective. Furthermore, in the range from 20 to 100 lines, data-flow required the inspection of less lines of code in comparison to control-flow with statistical significance. Data-flow, though, is more expensive than control-flow: it takes 23% to 245% longer to collect and rank the elements, on average data-flow is 129% more expensive.

Our hypothesis in this work was that data-flow is more effective in ranking the suspicious elements because it tracks more connections — more definition-use associations — at run-time. Our results suggest that such a hypothesis is true. However, data-flow coverage efficiency in SFL should be improved to become a common practice at industrial settings.

6.2 Contributions

This research gave origin to several contributions to SFL and specifically for SFL supported by data-flow coverage. They are described below:

- A literature review. We conducted a literature review on how data-flow information is used in SFL. We researched works that used data-flow information to support fault localization in Chapter 3. The research indicates that data-flow use in debugging is in its infancy, the techniques are mostly assessed with small to medium-sized programs, and do not report the time and memory overhead.
- The Jaguar tool. We developed an open source tool that integrates control- and data-flow coverage tools (JaCoCo and BA-DUA) and employ them to rank the most suspicious code elements by using SFL techniques. The tool has a user interface that collects JUnit information and control- and data-flow coverage to debug Java programs. It was also developed a graphic presentation of the elements (dua and lines) to facilitate experiments with users.
- Heuristics performance. An experiment assessed how ten different heuristics perform using control- and data-flow coverages. In general, Kulczynski2 and McCon performed better for small and mid-range effort budgets while Ochiai was superior for higher budgets.

- Data- and control-flow effectiveness comparison. Our data indicates that data-flow locates more bugs in small to mid-range budgets. A paired statistical comparison was conducted. Data-flow ranked the defects better than control-flow for budgets from 20 to 100 with a statistical significance level of 5%.
- Data- and control-flow efficiency comparison. The cost in terms of execution time to generate suspiciousness information from control- and data-flow coverages were compared. Data-flow is still more expensive than control-flow. To be used in industrial settings, data-flow coverage should be collected in such a way to be used in SFL techniques.

We believe that our main contribution is to provide guidance to the practitioner on when to use control- and data-flow coverages in SFL.

6.3 Future work

Coverage tools should support SFL techniques in native mode, which would save time and memory. For example, they could save the number of times a code element (statement, branch or dua) was executed for failing and passing tests. This change would reduce significantly the time spent on communication between the SFL tool (Jaguar) and coverage tools (e.g., JaCoCo and BA-DUA) and on coverage storage as well. We plan to address this issue in future versions of BA-DUA.

To enhance the effectiveness of data-flow coverage, BA-DUA should address its known limitations: single-block methods with no coverage, run-time exceptions coverage, and inter-procedural definition-use associations. Some of these enhancements are simple to implement and would improve significantly the results (e.g., single-block method coverage). Other features are more complex to implement and may imply a higher performance overhead, such as tracking run-time exceptions coverage and inter-procedural definition-use associations.

Our results should be backed up by user studies. One particular aspect to be investigated is how the variables associated to the most suspicious duas can be used to bring more insights to the developer while investigating the defect causes and consequences. Jaguar is prepared for such studies and we plan to conduct them with it in the future.

Bibliography

AGRAWAL, H.; HORGAN, J.; LONDON, S.; WONG, W. Fault localization using execution slices and dataflow tests. In: *Software Reliability Engineering, 1995. Proceedings., Sixth International Symposium on*. [S.l.: s.n.], 1995. p. 143–151. Citado 3 vezes nas páginas 41, 42, and 49.

ALVES, E.; GLIGORIC, M.; JAGANNATH, V.; D'AMORIM, M. Fault-localization using dynamic slicing and change impact analysis. In: *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2011. (ASE '11), p. 520–523. Citado 5 vezes nas páginas 39, 41, 44, 45, and 49.

ARAKI, K.; FURUKAWA, Z.; CHENG, J. A general framework for debugging. *IEEE Software Magazine*, v. 8, n. 3, p. 14–20, 1991. Citado na página 15.

ARAUJO, R. P. A. D.; CHAIM, M. L. Data-Flow Testing in the Large. *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, Ieee, p. 81–90, mar. 2014. Citado 5 vezes nas páginas 18, 24, 52, 72, and 75.

ASSI, R.; MASRI, W. Identifying failure-correlated dependence chains. In: *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*. [S.l.: s.n.], 2011. p. 607–616. Citado 4 vezes nas páginas 41, 42, 44, and 48.

BIOLCHINI, J.; MIAN, P. G.; NATALI, A. C. C.; TRAVASSOS, G. H. Systematic review in software engineering. *System Engineering and Computer Science Department COPPE/UFRJ, Technical Report ES*, v. 679, n. 05, p. 45, 2005. Citado na página 34.

CAO, H.; JIANG, S.; JU, X.; ZHANG, Y.; YUAN, G. Applying association analysis to dynamic slicing based fault localization. In: _____. [S.l.: s.n.], 2014. E97-D, n. 8, p. 2057–2066. Citado 2 vezes nas páginas 40 and 47.

CHAIM, M.; MALDONADO, J.; JINO, M. A debugging strategy based on requirements of testing. In: *Software Maintenance and Reengineering, 2003. Proceedings. Seventh European Conference on*. [S.l.: s.n.], 2003. p. 160–169. Citado 5 vezes nas páginas 39, 42, 44, 45, and 49.

CHAIM, M. L.; ARAUJO, R. P. A. de. An efficient bitwise algorithm for intra-procedural data-flow testing coverage. *Information Processing Letters*, v. 113, n. 8, p. 293 – 300, 2013. Citado 6 vezes nas páginas 22, 23, 25, 26, 27, and 28.

CHAIM, M. L.; MALDONADO, J.; JINO, M. A debugging strategy based on requirements of testing. *Seventh European Conference on Software Maintenance and Reengineering, 2003. Proceedings.*, p. 1–31, 2003. Citado 3 vezes nas páginas 14, 18, and 24.

DANDAN, G.; TIAN, W.; XIAOHONG, S.; PEIJUN, M.; YU, W. State Dependency Probabilistic Model for Fault Localization. *Information and Software Technology*, Elsevier B.V., jun. 2014. Citado 2 vezes nas páginas 14 and 24.

DELAMARO, M. E.; CHAIM, M. L.; VINCENZI, A. M. R. Técnicas e ferramentas de teste de software. In: *Atualizações em Informática 2010 (JAI 2010)*. [S.l.]: Editora PUC-Rio, 2010. p. 55–110. Citado na página 16.

EICHINGER, F.; KROGMANN, K.; KLUG, R.; BÖHM, K. Software-defect localisation by mining dataflow-enabled call graphs. In: *Proceedings of the 2010 European Conference on Machine Learning and Knowledge Discovery in Databases: Part I*. Berlin, Heidelberg: Springer-Verlag, 2010. (ECML PKDD'10), p. 425–441. Citado 4 vezes nas páginas 41, 42, 48, and 49.

FEITELSON, D. G.; FRACHTENBERG, E.; BECK, K. L. Development and deployment at facebook. *IEEE Internet Computing*, v. 17, n. 4, p. 8–17, jul. 2013. Citado na página 14.

HE, H.; ZHANG, D.; LIU, M.; ZHANG, W.; GAO, D. A coverage and slicing dependencies analysis for seeking software security defects. In: _____. [S.l.: s.n.], 2014. v. 2014. Citado 3 vezes nas páginas 40, 42, and 47.

HECHT, M. S. *Flow Analysis of Computer Programs*. New York, NY, USA: Elsevier Science Inc., 1977. Citado na página 25.

HOFER, B.; WOTAWA, F. *Spectrum enhanced dynamic slicing for better fault localization*. [S.l.: s.n.], 2012. v. 242. 420-425 p. (Frontiers in Artificial Intelligence and Applications, v. 242). Citado 4 vezes nas páginas 40, 42, 44, and 47.

HUIZINGA, D.; KOLAWA, A. Principles of automated defect prevention. In: _____. *Automated Defect Prevention*. [S.l.]: John Wiley & Sons, Inc., 2007. p. 19–51. Citado na página 22.

IEEE Standard Glossary of Software Engineering Terminology. *IEEE Std 610.12-1990*, p. 1–84, Dec 1990. Citado na página 22.

JONES, J.; HARROLD, M.; STASKO, J. Visualization of test information to assist fault localization. In: *International Conference on Software Engineering*. [S.l.]: Acm, 2002. p. 467–477. Citado 2 vezes nas páginas 17 and 31.

JONES, J. A.; BOWRING, J. F.; HARROLD, M. J. Debugging in parallel. In: *Proceedings of the 2007 International Symposium on Software Testing and Analysis*. [S.l.: s.n.], 2007. (ISSTA '07), p. 16–26. Citado na página 74.

JU, X.; JIANG, S.; CHEN, X.; WANG, X.; ZHANG, Y.; CAO, H. HSFal: Effective fault localization using hybrid spectrum of full slices and execution slices. *Journal of Systems and Software*, Elsevier Inc., v. 90, n. 1, p. 3–17, abr. 2014. Citado 3 vezes nas páginas 29, 30, and 31.

JU, X.; JIANG, S.; CHEN, X.; WANG, X.; ZHANG, Y.; CAO, H. Hsfal: Effective fault localization using hybrid spectrum of full slices and execution slices. *Journal of Systems and Software*, v. 90, n. 0, p. 3 – 17, 2014. Citado 4 vezes nas páginas 39, 41, 44, and 46.

JUST, R.; JALALI, D.; ERNST, M. D. Defects4J: A Database of existing faults to enable controlled testing studies for Java programs. In: *ISSTA 2014, Proceedings of the 2014 International Symposium on Software Testing and Analysis*. San Jose, CA, USA: [s.n.], 2014. p. 437–440. Tool demo. Citado na página 59.

KITCHENHAM, B. Procedures for performing systematic reviews. *Keele, UK, Keele University*, v. 33, p. 2004, 2004. Citado na página 34.

KOREL, B.; LASKI, J. Dynamic program slicing. *Information Processing Letters*, v. 29, n. 3, p. 155–163, 1988. Citado na página 16.

LAWRANCE, J.; BOGART, C. How programmers debug, revisited: An information foraging theory perspective. v. 39, n. 2, p. 197–215, 2013. Citado na página 15.

LEI, Y.; MAO, X.; DAI, Z.; WANG, C. Effective statistical fault localization using program slices. In: *Computer Software and Applications Conference (COMPSAC), 2012 IEEE 36th Annual*. [S.l.: s.n.], 2012. p. 1–10. Citado 4 vezes nas páginas 40, 42, 44, and 45.

LIU, Y.; LI, W.; JIANG, S.; ZHANG, Y.; JU, X. An approach for fault localization based on program slicing and bayesian. In: *Quality Software (QSIC), 2013 13th International Conference on*. [S.l.: s.n.], 2013. p. 326–332. Citado 4 vezes nas páginas 40, 41, 47, and 49.

MA, P.; WANG, Y.; SU, X.; WANG, T. A novel fault localization method with fault propagation context analysis. In: *Instrumentation, Measurement, Computer, Communication and Control (IMCCC), 2013 Third International Conference on*. [S.l.: s.n.], 2013. p. 1194–1199. Citado 2 vezes nas páginas 40 and 47.

MAO, X.; LEI, Y.; DAI, Z.; QI, Y.; WANG, C. Slice-based statistical fault localization. *Journal of Systems and Software*, Elsevier Inc., v. 89, n. 1, p. 51–62, mar. 2014. Citado 7 vezes nas páginas 14, 17, 18, 24, 29, 30, and 31.

MAO, X.; LEI, Y.; DAI, Z.; QI, Y.; WANG, C. Slice-based statistical fault localization. *Journal of Systems and Software*, v. 89, n. 0, p. 51 – 62, 2014. Citado 5 vezes nas páginas 39, 41, 43, 44, and 45.

MASRI, W. Fault localization based on information flow coverage. *Software Testing, Verification and Reliability*, John Wiley & Sons, Ltd., v. 20, n. 2, p. 121–147, 2010. Citado 4 vezes nas páginas 39, 42, 44, and 46.

MOIR, K. Releng of the nerds: Open source release engineering. SDK code coverage with JaCoCo. 2011. Disponível em: <http://relengofthenerds.blogspot.com.br/2011/03/sdk-code-coverage-with-jacoco.html>. Citado na página 75.

PARNIN, C.; ORSO, A. Are automated debugging techniques actually helping programmers? In: *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. [S.l.: s.n.], 2011. (ISSTA '11), p. 199–209. Citado na página 74.

RAPPS, S.; WEYUKER, E. Selecting software test data using data flow information. *Software Engineering, IEEE Transactions on*, SE-11, n. 4, p. 367–375, April 1985. Citado 2 vezes nas páginas 17 and 28.

SANTELICES, R.; JONES, J. A.; HARROLD, M. J. Lightweight fault-localization using multiple coverage types. In: *2009 IEEE 31st International Conference on Software Engineering*. [S.l.: IEEE, 2009. p. 56–66. Citado 2 vezes nas páginas 17 and 18.

SANTELICES, R.; JONES, J. A.; YU, Y.; HARROLD, M. J. Lightweight fault-localization using multiple coverage types. In: *Proceedings of the 31st International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2009. (ICSE '09), p. 56–66. Citado 3 vezes nas páginas 39, 42, and 45.

- SOUZA, H. A. de. *Depuração de programas baseada em cobertura de integração*. 148 p. Tese (Doutorado) — Universidade de São Paulo, 2012. Disponível em: <http://www.teses.usp.br/teses/disponiveis/100/100131/tde-08032013-162246/en.php>. Citado 3 vezes nas páginas 26, 32, and 33.
- STALLMAN, R.; PESCH, R. *Debugging with GDB: The GNU Source-level Debugger*. [S.l.]: Free Software Foundation, 1992. Citado na página 16.
- SUN, J. .; LI, Z. .; NI, J. . *Dichotomy method toward interactive testing-based fault localization*. [S.l.: s.n.], 2008. v. 5139 LNAI. 182-193 p. (Lecture notes in Computer Science (including subseries Lecture notes in Artificial Intelligence and Lecture notes in Bioinformatics), v. 5139 LNAI). Citado 4 vezes nas páginas 41, 42, 48, and 49.
- SUN, J.; LI, Z.; NI, J.; YIN, F. Software fault localization based on testing requirement and program slice. In: *Networking, Architecture, and Storage, 2007. NAS 2007. International Conference on*. [S.l.: s.n.], 2007. p. 168–176. Citado 4 vezes nas páginas 41, 42, 48, and 49.
- WANG, T.; ROYCHOUDHURY, A. Hierarchical dynamic slicing. In: *Proceedings of the 2007 International Symposium on Software Testing and Analysis*. New York, NY, USA: ACM, 2007. (ISSTA '07), p. 228–238. Citado 3 vezes nas páginas 41, 42, and 48.
- WEISER, M. Program slicing. In: *Proceedings of the 5th International Conference on Software Engineering*. [S.l.]: IEEE Press, 1981. (ICSE '81), p. 439–449. Citado na página 16.
- WEN, W.; LI, B.; SUN, X.; LI, J. Program slicing spectrum-based software fault localization. In: *SEKE 2011 - Proceedings of the 23rd International Conference on Software Engineering and Knowledge Engineering*. [S.l.: s.n.], 2011. p. 213–218. Citado 3 vezes nas páginas 39, 41, and 46.
- WONG, W.; QI, Y. An execution slice and inter-block data dependency-based approach for fault localization. In: *Software Engineering Conference, 2004. 11th Asia-Pacific*. [S.l.: s.n.], 2004. p. 366–373. Citado 3 vezes nas páginas 41, 42, and 48.
- WONG, W. E.; QI, Y. Effective program debugging based on execution slices and inter-block data dependency. *Journal of Systems and Software*, v. 79, n. 7, p. 891 – 903, 2006. Citado 3 vezes nas páginas 41, 42, and 48.
- XU, X.; DEBROY, V.; WONG, W. E.; GUO, D. Ties within fault localization rankings: Exposing and addressing the problem. In: _____. [S.l.: s.n.], 2011. v. 21, n. 6, p. 803–827. Citado 2 vezes nas páginas 41 and 42.
- YANG, B.; WU, J.; LIU, C. Mining data chain graph for fault localization. In: *Computer Software and Applications Conference Workshops (COMPSACW), 2012 IEEE 36th Annual*. [S.l.: s.n.], 2012. p. 464–469. Citado 4 vezes nas páginas 40, 42, 47, and 49.
- YOU, Y.-S.; HUANG, C.-Y.; PENG, K.-L.; HSU, C.-J. Evaluation and Analysis of Spectrum-Based Fault Localization with Modified Similarity Coefficients for Software Debugging. *2013 IEEE 37th Annual Computer Software and Applications Conference*, Ieee, p. 180–189, jul. 2013. Citado na página 24.

YU, R.; ZHAO, L.; WANG, L.; YIN, X. Statistical fault localization via semi-dynamic program slicing. In: *Trust, Security and Privacy in Computing and Communications (TrustCom), 2011 IEEE 10th International Conference on*. [S.l.: s.n.], 2011. p. 695–700. Citado 4 vezes nas páginas 40, 42, 44, and 48.

ZELLER, A. *Why Programs Fail: A Guide to Systematic Debugging*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005. Citado 2 vezes nas páginas 22 and 24.

ZHANG, L.; KIM, M.; KHURSHID, S. Faulttracer: A spectrum-based approach to localizing failure-inducing program edits. In: _____. [S.l.: s.n.], 2013. v. 25, n. 12, p. 1357–1383. Citado 5 vezes nas páginas 40, 42, 44, 47, and 49.

ZHANG, Z.; MAO, X.; LEI, Y.; ZHANG, P. Enriching contextual information for fault localization. In: _____. [S.l.: s.n.], 2014. E97-D, n. 6, p. 1652–1655. Citado 3 vezes nas páginas 39, 41, and 46.

APPENDIX A – Research Strings

ACM:

(
 Title: “fault-localization” OR Title: “fault-localisation” OR
 Title: “defect-localisation” OR Title: “defect-localization” OR
 Title: “fault localization” OR Title: “fault localisation” OR
 Title: “defect localisation” OR Title: “defect localization” OR
 Title: “SFL” OR Title: “SBFL” OR Title: “CBFL” OR
 Abstract: “fault-localization” OR Abstract: “fault-localisation” OR
 Abstract: “defect-localisation” OR Abstract: “defect-localization” OR
 Abstract: “fault localization” OR Abstract: “fault localisation” OR
 Abstract: “defect localisation” OR Abstract: “defect localization” OR
 Abstract: “SFL” OR Abstract: “SBFL” OR Abstract: “CBFL”
)
 AND
 (
 Title: “slice” OR Title: “slicing” OR
 Title: “dua” OR Title: “def-use” OR
 Title: “du-pair” OR Title: “du-pairs” OR
 Title: “definition-use” OR
 Title: “data-flow” OR Title: “data flow” OR Title: “dataflow” OR
 Title: “information-flow” OR Title: “information flow” OR
 Title: “data dependency” OR Title: “data dependencies” OR
 Abstract: “slice” OR Abstract: “slicing” OR
 Abstract: “dua” OR Abstract: “def-use” OR
 Abstract: “du-pair” OR Abstract: “du-pairs” OR
 Abstract: “definition-use” OR
 Abstract: “data-flow” OR Abstract: “data flow” OR Abstract: “dataflow” OR
 Abstract: “information-flow” OR Abstract: “information flow” OR
 Abstract: “data dependency” OR Abstract: “data dependencies”
)

IEEE: (

“fault localization” OR

“fault localisation” OR

“defect localisation” OR

“defect localization” OR

“SFL” OR

“SBFL” OR

“CBFL”

)

AND

(

“slice” OR

“slicing” OR

“dua” OR

“def-use” OR

“du-pair” OR

“definition-use” OR

“data dependencies” OR

“data dependency” OR

“definition-use” OR

“data flow” OR

“information flow”)

CAPES:

1) (fault localization OR localização de falha)

2) (defect localization OR localização de defeito)

USP:

(localização de defeito OR localização de falha)

Wiley:

(

“fault localization” OR

“fault localisation” OR

“defect localisation” OR
 “defect localization” OR
 “SFL” OR
 “SBFL” OR
 “CBFL”
)
 AND
 (
 “slice” OR
 “slicing” OR
 “dua” OR
 “def-use” OR
 “du-pair” OR
 “du-pairs” OR
 “definition-use” OR
 “data-flow” OR
 “data flow” OR
 “dataflow” OR
 “information-flow” OR
 “information flow” OR
 “data dependency” OR
 “data dependencies”)

Science Direct:

(
 TITLE-ABSTR-KEY(“fault localization”) OR
 TITLE-ABSTR-KEY(“fault-localization”) OR
 TITLE-ABSTR-KEY(“fault localisation”) OR
 TITLE-ABSTR-KEY(“fault-localisation”) OR
 TITLE-ABSTR-KEY(“defect localisation”) OR
 TITLE-ABSTR-KEY(“defect-localisation”) OR
 TITLE-ABSTR-KEY(“defect localization”) OR
 TITLE-ABSTR-KEY(“defect-localization”) OR
 TITLE-ABSTR-KEY(“SFL”) OR

TITLE-ABSTR-KEY("SBFL") OR
 TITLE-ABSTR-KEY("CBFL")
)
 AND
 (
 TITLE-ABSTR-KEY("slice") OR
 TITLE-ABSTR-KEY("slicing") OR
 TITLE-ABSTR-KEY("dua") OR
 TITLE-ABSTR-KEY("def-use") OR
 TITLE-ABSTR-KEY("du-pair") OR
 TITLE-ABSTR-KEY("du-pairs") OR
 TITLE-ABSTR-KEY("definition-use") OR
 TITLE-ABSTR-KEY("data-flow") OR
 TITLE-ABSTR-KEY("data flow") OR
 TITLE-ABSTR-KEY("dataflow") OR
 TITLE-ABSTR-KEY("information-flow") OR
 TITLE-ABSTR-KEY("information flow") OR
 TITLE-ABSTR-KEY("data dependency") OR
 TITLE-ABSTR-KEY("data dependencies")
)

Scopus:

(
 TITLE-ABS-KEY("fault localization") OR
 TITLE-ABS-KEY("fault-localization") OR
 TITLE-ABS-KEY("fault localisation") OR
 TITLE-ABS-KEY("fault-localisation") OR
 TITLE-ABS-KEY("defect localisation") OR
 TITLE-ABS-KEY("defect-localisation") OR
 TITLE-ABS-KEY("defect localization") OR
 TITLE-ABS-KEY("defect-localization") OR
 TITLE-ABS-KEY("SFL") OR
 TITLE-ABS-KEY("SBFL") OR

```

TITLE-ABS-KEY("CBFL")
)
AND
(
TITLE-ABS-KEY("slice") OR
TITLE-ABS-KEY("slicing") OR
TITLE-ABS-KEY("dua") OR
TITLE-ABS-KEY("def-use") OR
TITLE-ABS-KEY("du-pair") OR
TITLE-ABS-KEY("du-pairs") OR
TITLE-ABS-KEY("definition-use") OR
TITLE-ABS-KEY("data-flow") OR
TITLE-ABS-KEY("data flow") OR
TITLE-ABS-KEY("dataflow") OR
TITLE-ABS-KEY("information-flow") OR
TITLE-ABS-KEY("information flow") OR
TITLE-ABS-KEY("data dependency") OR
TITLE-ABS-KEY("data dependencies")
)

```


APPENDIX B – Heuristic versus heuristics: statistical tests for control- and data-flow coverages

We applied the paired Wilcoxon test among heuristics for the seven budgets and for each coverage (control- and data-flow). In what follows, we present the p-values obtained in the statistical tests carried out.

B.1 Heuristic versus heuristic: Control-flow

The p-values obtained in the comparison of the heuristics using control-flow for each budget are presented in Table 15 (budget 5), Table 16 (budget 10), Table 17 (budget 20), Table 18 (budget 30), Table 19 (budget 40), Table 20 (budget 50), and Table 21 (budget 100). The significant p-values are printed in boldface.

Table 15 – Heuristic versus heuristic — Control-flow — Budget 5

Heuristic	DRT	Jaccard	Kulcz.	Mccon	Minus	Ochiai	Op	Taran.	Wong3	Zoltar
DRT	-	42.8	10.2	10.2	97.7	36.2	100	82.0	97.0	14.5
Jaccard	61.7	-	4.8	4.8	75.9	50.0	61.7	92.4	98.5	20.8
Kulcz.	92.0	97.8	-	100	97.2	97.1	92.0	97.9	99.9	81.4
Mccon	92.0	97.8	100	-	97.2	97.1	92.0	97.9	99.9	81.4
Minus	50.0	28.6	3.8	3.8	-	22.2	50.0	73.7	95.4	4.4
Ochiai	68.9	97.7	8.6	8.6	82.5	-	68.9	94.9	99.1	29.0
Op	100	42.8	10.2	10.2	97.7	36.2	-	82.0	97.0	14.5
Taran.	19.6	9.4	2.4	2.4	28.6	6.1	19.6	-	83.7	4.4
Wong3	3.3	1.6	0.06	0.06	5.1	0.9	3.3	17.1	-	0.06
Zoltar	89.7	86.0	50.0	50.0	97.1	82.1	89.7	96.2	99.9	-

Source: Henrique Ribeiro, 2016

For control-flow and budget 5, Kulczynski2 and Mccon are significantly better ($p - value \leq 5\%$) than four other heuristics (Jaccard, Minus, Tarantula and Wong3) and Wong3 is worst than all of them (except for Tarantula) also with significance level of 5%.

For control-flow and budget 10, Kulczynski2 and Mccon are significantly better ($p - value \leq 5\%$) than four other heuristics (Jaccard, Minus, Tarantula and Wong3) and Wong3 is worst than all of them also with significance level of 5%. Occhiai is significantly better than Tarantula for this budget.

For control-flow and budget 20, Jaccard is significantly better ($p - value \leq 5\%$) than three other heuristics (Minus, Tarantula and Wong3) and Wong3 is worst than all of them also with significance level of 5%.

Table 16 – Heuristic versus heuristic — Control-flow — Budget 10

Heuristic	DRT	Jaccard	Kulcz.	Mccon	Minus	Ochiai	Op	Taran.	Wong3	Zoltar
DRT	-	33.1	13.7	13.7	96.3	25.4	100	81.6	99.9	18.2
Jaccard	68.0	-	4.52	4.52	85.2	28.5	68.0	95.4	100	17.0
Kulcz.	87.2	96.6	-	100	97.2	88.4	87.2	98.8	100	81.4
Mccon	87.2	96.6	100	-	97.2	88.4	87.2	98.8	100	81.4
Minus	18.5	15.6	3.01	3.01	-	10.2	18.5	69.1	99.9	4.11
Ochiai	75.7	82.7	14.5	14.5	90.5	-	75.7	97.6	100	31.7
Op	100	33.1	13.7	13.7	96.3	25.4	-	81.6	99.9	18.2
Taran.	18.9	5.36	1.24	1.24	31.8	2.75	18.9	-	99.7	2.38
Wong3	0.01	0.001	0.001	0.001	0.03	0.001	0.01	0.28	-	0.001
Zoltar	83.1	85.7	50.0	50.0	96.3	72.3	83.1	97.8	100	-

Source: Henrique Ribeiro, 2016

Table 17 – Heuristic versus heuristic — Control-flow — Budget 20

Heuristic	DRT	Jaccard	Kulcz.	Mccon	Minus	Ochiai	Op	Taran.	Wong3	Zoltar
DRT	-	7.57	15.8	15.8	81.9	15.8	100	45.0	100	22.2
Jaccard	92.6	-	74.6	74.6	97.0	78.3	92.6	97.3	100	84.8
Kulcz.	84.8	26.2	-	100	94.1	42.8	84.8	88.2	100	90.9
Mccon	84.8	26.2	100	-	94.1	42.8	84.8	88.2	100	90.9
Minus	29.1	3.0	6.2	6.2	-	8.5	29.1	34.6	100	11.1
Ochiai	84.5	23.8	58.3	58.3	91.7	-	84.5	86.7	100	74.2
Op	100	7.5	15.8	15.8	81.9	15.8	-	45.0	100	22.2
Taran.	55.3	2.7	11.9	11.9	65.7	13.6	55.3	-	100	20.6
Wong3	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	-	0.001
Zoltar	78.7	15.7	21.1	21.1	89.3	26.7	78.7	79.6	100	-

Source: Henrique Ribeiro, 2016

Table 18 – Heuristic versus heuristic — Control-flow — Budget 30

Heuristic	DRT	Jaccard	Kulcz.	Mccon	Minus	Ochiai	Op	Taran.	Wong3	Zoltar
DRT	-	4.3	10.8	10.8	60.6	3.5	97.7	20.9	100	18.7
Jaccard	95.7	-	81.5	81.5	97.8	46.7	95.8	95.9	100	87.0
Kulcz.	89.5	18.9	-	100	94.7	24.7	89.5	68.1	100	81.9
Mccon	89.5	18.9	100	-	94.7	24.7	89.5	68.1	100	81.9
Minus	50.0	2.2	5.5	5.5	-	1.5	50.0	15.9	100	10.5
Ochiai	96.5	54.8	76.0	76.0	98.4	-	96.6	94.8	100	85.8
Op	50.0	4.2	10.8	10.8	60.6	3.4	-	20.9	100	15.5
Taran.	79.2	4.2	32.2	32.2	84.2	5.3	79.2	-	100	43.0
Wong3	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	-	0.001
Zoltar	82.0	13.3	29.1	29.1	89.8	14.7	85.1	57.3	100	-

Source: Henrique Ribeiro, 2016

For control-flow and budget 30, Jaccard is significantly better ($p - value \leq 5\%$) than five other heuristics (DRT, Minus, OP, Tarantula and Wong3), Ochiai is better than four other heuristics (DRT, Minus, OP and Wong3) and Wong3 is worst than all of them also with significance level of 5%.

Table 19 – Heuristic versus heuristic — Control-flow — Budget 40

Heuristic	DRT	Jaccard	Kulcz.	Mecon	Minus	Ochiai	Op	Taran.	Wong3	Zoltar
DRT	-	2.8	8.1	8.1	60.6	2.5	97.7	16.3	100	14.8
Jaccard	97.1	-	81.9	81.9	98.2	48.7	97.4	93.7	100	87.2
Kulcz.	92.2	18.5	-	100	96.1	28.2	92.3	64.9	100	81.9
Mecon	92.2	18.5	100	-	96.1	28.2	92.3	64.9	100	81.9
Minus	50.0	1.8	4.0	4.0	-	1.3	50.0	12.9	100	9.31
Ochiai	97.5	52.6	72.4	72.4	98.6	-	97.8	93.7	100	82.2
Op	50.0	2.5	7.9	7.9	60.6	2.2	-	15.4	100	11.8
Taran.	83.8	6.4	35.4	35.4	87.2	6.4	84.7	-	100	46.4
Wong3	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	-	0.001
Zoltar	85.7	13.0	29.1	29.1	91.0	18.2	88.7	53.9	100	-

Source: Henrique Ribeiro, 2016

For control-flow and budget 40, Jaccard and Ochiai are significantly better ($p - value \leq 5\%$) than four other heuristics (DRT, Minus, OP and Wong3) and Wong3 is worst than all of them also with significance level of 5%.

Table 20 – Heuristic versus heuristic — Control-flow — Budget 50

Heuristic	DRT	Jaccard	Kulcz.	Mecon	Minus	Ochiai	Op	Taran.	Wong3	Zoltar
DRT	-	1.6	7.2	7.2	58.3	1.2	97.7	13.1	100	15.7
Jaccard	98.4	-	86.2	86.2	99.1	36.5	98.6	95.9	100	89.8
Kulcz.	93.0	14.0	-	100	96.6	18.5	93.3	59.7	100	81.9
Mecon	93.0	14.0	100	-	96.6	18.5	93.3	59.7	100	81.9
Minus	50.0	0.8	3.4	3.4	-	0.6	50.0	10.2	100	8.8
Ochiai	98.7	64.5	82.0	82.0	99.3	-	98.9	94.0	100	88.9
Op	50.0	1.3	6.9	6.9	58.3	1.0	-	12.2	100	10.5
Taran.	86.9	4.2	40.6	40.6	89.9	6.0	87.8	-	100	50.9
Wong3	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	-	0.001
Zoltar	84.8	10.3	29.1	29.1	91.5	11.4	89.9	49.4	100	-

Source: Henrique Ribeiro, 2016

For control-flow and budget 50, Jaccard is significantly better ($p - value \leq 5\%$) than five other heuristics (DRT, Minus, OP, Tarantula and Wong3), Ochiai is better than four other heuristics (DRT, Minus, OP and Wong3) and Wong3 is worst than all of them also with significance level of 5%.

Table 21 – Heuristic versus heuristic - Control-flow - Budget 100

Heuristic	DRT	Jaccard	Kulcz.	Mccon	Minus	Ochiai	Op	Taran.	Wong3	Zoltar
DRT	-	0.3	11.8	11.8	45.2	0.1	97.7	5.1	100	21.0
Jaccard	99.6	-	93.6	93.6	99.6	13.5	99.6	89.9	100	95.4
Kulcz.	88.4	6.5	-	100	93.7	3.1	88.8	39.5	100	70.5
Mccon	88.4	6.5	100	-	93.7	3.1	88.8	39.5	100	70.5
Minus	59.4	0.3	6.47	6.4	-	0.09	63.9	5.3	100	13.4
Ochiai	99.8	86.8	96.9	96.9	99.9	-	99.9	96.0	100	98.2
Op	50.0	0.3	11.5	11.5	40.6	0.1	-	4.7	100	17.5
Taran.	94.8	10.3	60.7	60.7	94.7	4.0	95.3	-	100	69.3
Wong3	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	-	0.001
Zoltar	79.4	4.6	39.3	39.3	86.8	1.8	82.8	30.8	100	-

Source: Henrique Ribeiro, 2016

For control-flow and budget 100, Ochiai is better than all other heuristics (except for Jaccard) and Wong3 is worst than all of them also with significance level of 5%.

B.2 Heuristic versus heuristic: Data-flow

The p-values obtained in the comparison of the heuristics using data-flow for each budget are presented in Table 22 (budget 5), Table 23 (budget 10), Table 24 (budget 20), Table 25 (budget 30), Table 26 (budget 40), Table 27 (budget 50), and Table 28 (budget 100). The significant p-values are printed in boldface.

Table 22 – Heuristic versus heuristic — Data-flow — Budget 5

Heuristic	DRT	Jaccard	Kulcz.	Mccon	Minus	Ochiai	Op	Taran.	Wong3	Zoltar
DRT	-	91.0	75.1	75.1	100	84.7	100	99.0	99.3	95.1
Jaccard	11.4	-	8.6	8.6	11.4	50.0	11.4	97.8	78.2	35.5
Kulcz.	34.1	97.1	-	100	34.1	97.0	34.1	99.1	93.2	97.7
Mccon	34.1	97.1	100	-	34.1	97.0	34.1	99.1	93.2	97.7
Minus	100	91.0	75.1	75.1	-	84.7	100	99.0	99.3	95.1
Ochiai	19.6	97.7	17.2	17.2	19.6	-	19.6	98.6	89.8	60.7
Op	100	91.0	75.1	75.1	100	84.7	-	99.0	99.3	95.1
Taran.	1.4	7.4	1.5	1.5	1.4	3.5	1.4	-	35.1	2.6
Wong3	0.9	23.9	8.1	8.1	0.9	11.6	0.9	67.2	-	14.0
Zoltar	9.8	77.1	50.0	50.0	9.8	60.7	9.8	98.6	88.4	-

Source: Henrique Ribeiro, 2016

For Data-flow and budget 5, DRT, Minus and Op are better than two other heuristics (Wong3 and Tarantula) with significance level of 5%.

Table 23 – Heuristic versus heuristic — Data-flow — Budget 10

Heuristic	DRT	Jaccard	Kulcz.	Mccon	Minus	Ochiai	Op	Taran.	Wong3	Zoltar
DRT	-	68.8	32.2	32.2	50.0	70.3	97.7	99.0	99.1	52.0
Jaccard	32.9	-	7.0	7.0	28.4	50.0	40.7	99.5	95.7	27.6
Kulcz.	71.3	95.3	-	100	63.9	95.1	76.2	99.9	99.5	97.7
Mccon	71.3	95.3	100	-	63.9	95.1	76.2	99.9	99.5	97.7
Minus	97.7	73.4	40.6	40.6	-	75.8	96.3	99.2	99.4	59.4
Ochiai	31.8	81.4	9.8	9.8	26.3	-	41.2	99.7	96.3	39.3
Op	50.0	61.2	27.0	27.0	18.5	61.1	-	98.7	98.8	42.9
Taran.	1.0	0.6	0.09	0.09	0.8	0.3	1.4	-	47.9	0.1
Wong3	0.8	4.5	0.5	0.5	0.6	3.9	1.2	52.8	-	0.8
Zoltar	52.0	77.7	50.0	50.0	45.2	70.6	61.6	99.8	99.2	-

Source: Henrique Ribeiro, 2016

For data-flow and budget 10, eight heuristics are better than the two other (Wong3 and Tarantula) with significance level of 5%.

Table 24 – Heuristic versus heuristic — Data-flow — Budget 20

Heuristic	DRT	Jaccard	Kulcz.	Mccon	Minus	Ochiai	Op	Taran.	Wong3	Zoltar
DRT	-	23.1	2.7	2.7	50.0	11.5	97.7	87.9	99.9	12.6
Jaccard	77.5	-	16.4	16.4	74.9	10.3	81.9	99.5	99.9	38.8
Kulcz.	97.5	85.0	-	100	97.1	74.8	98.2	99.5	100	97.1
Mccon	97.5	85.0	100	-	97.1	74.8	98.2	99.5	100	97.1
Minus	97.7	25.8	3.2	3.2	-	12.9	96.3	89.1	99.9	14.1
Ochiai	88.9	92.9	27.7	27.7	87.6	-	91.9	99.8	100	50.0
Op	50.0	18.6	1.9	1.9	18.5	8.4	-	82.9	99.9	5.3
Taran.	12.4	0.5	0.4	0.4	11.2	0.1	17.6	-	98.3	1.1
Wong3	0.05	0.02	0.001	0.001	0.06	0.003	0.08	1.6	-	0.001
Zoltar	88.4	63.3	8.67	8.67	87.2	52.7	95.2	98.9	100	-

Source: Henrique Ribeiro, 2016

For data-flow and budget 20, Kulczynski2 and Mccon are significantly better ($p - value \leq 5\%$) than five other heuristics (DRT, Minus, OP, Tarantula and Wong3) and Wong3 is worst than all of them also with significance level of 5%.

For Data-flow and budget 30, Kulczynski2, Mccon and Ochiai are significantly better ($p - value \leq 5\%$) than five other heuristics (DRT, Minus, OP, Tarantula and Wong3) and Wong3 is worst than all of them also with significance level of 5%.

For data-flow and budget 40, Kulczynski2, Mccon and Ochiai is significantly better ($p - value \leq 5\%$) than five other heuristics (DRT, Minus, OP, Tarantula and Wong3) and Wong3 is worst than all of them also with significance level of 5%.

Table 25 – Heuristic versus heuristic — Data-flow — Budget 30

Heuristic	DRT	Jaccard	Kulcz.	Mccon	Minus	Ochiai	Op	Taran.	Wong3	Zoltar
DRT	-	18.3	0.8	0.8	17.2	4.37	97.7	65.2	100	6.1
Jaccard	82.1	-	24.6	24.6	79.1	7.6	86.1	99.5	100	49.1
Kulcz.	99.2	76.8	-	100	98.9	64.7	99.4	98.5	100	95.1
Mccon	99.2	76.8	100	-	98.9	64.7	99.4	98.5	100	95.1
Minus	97.0	21.4	1.1	1.1	-	5.0	97.1	67.4	100	7.7
Ochiai	95.8	93.8	37.6	37.6	95.2	-	97.1	99.9	100	74.3
Op	50.0	14.2	0.6	0.6	8.6	3.0	-	55.8	100	2.1
Taran.	35.3	0.5	1.5	1.5	33.1	0.1	44.7	-	99.9	5.2
Wong3	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.006	-	0.001
Zoltar	94.3	52.6	9.87	9.87	92.9	27.5	98.0	94.9	100	-

Source: Henrique Ribeiro, 2016

Table 26 – Heuristic versus heuristic — Data-flow — Budget 40

Heuristic	DRT	Jaccard	Kulcz.	Mccon	Minus	Ochiai	Op	Taran.	Wong3	Zoltar
DRT	-	7.4	0.3	0.3	17.2	2.9	97.7	50.8	100	4.6
Jaccard	92.8	-	32.4	32.4	91.9	11.4	94.7	99.5	100	66.5
Kulcz.	99.6	68.7	-	100	99.5	57.9	99.7	97.1	100	95.1
Mccon	99.6	68.7	100	-	99.5	57.9	99.7	97.1	100	95.1
Minus	97.0	8.3	0.4	0.4	-	3.3	97.1	52.4	100	5.3
Ochiai	97.1	90.1	43.5	43.5	96.7	-	98.0	99.5	100	76.1
Op	50.0	5.3	0.2	0.2	8.6	2.0	-	40.8	100	1.4
Taran.	49.7	0.5	2.9	2.9	48.1	0.4	59.7	-	100	9.6
Wong3	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	-	0.001
Zoltar	95.8	34.6	9.8	9.8	95.1	25.0	98.7	90.6	100	-

Source: Henrique Ribeiro, 2016

Table 27 – Heuristic versus heuristic — Data-flow — Budget 50

Heuristic	DRT	Jaccard	Kulcz.	Mccon	Minus	Ochiai	Op	Taran.	Wong3	Zoltar
DRT	-	6.0	0.5	0.5	7.4	2.2	97.7	42.8	100	5.0
Jaccard	94.1	-	44.8	44.8	93.9	5.3	95.9	99.3	100	78.1
Kulcz.	99.5	56.4	-	100	99.4	48.5	99.6	95.2	100	91.2
Mccon	99.5	56.4	100	-	99.4	48.5	99.6	95.2	100	91.2
Minus	97.8	6.24	0.6	0.6	-	2.3	98.1	43.6	100	5.8
Ochiai	97.8	95.4	52.9	52.9	97.7	-	98.5	99.7	100	82.6
Op	50.0	4.2	0.3	0.3	4.4	1.5	-	33.1	100	1.7
Taran.	57.7	0.7	4.9	4.9	56.9	0.2	67.3	-	100	13.9
Wong3	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	-	0.001
Zoltar	95.3	22.7	13.9	13.9	94.6	18.2	98.4	86.4	100	-

Source: Henrique Ribeiro, 2016

For data-flow and budget 50, Kulczynski2, Mccon and Ochiai is significantly better ($p - value \leq 5\%$) than five other heuristics (DRT, Minus, OP, Tarantula and Wong3) and Wong3 is worst than all of them also with significance level of 5%.

Table 28 – Heuristic versus heuristic — Data-flow — Budget 100

Heuristic	DRT	Jaccard	Kulcz.	Mccon	Minus	Ochiai	Op	Taran.	Wong3	Zoltar
DRT	-	2.5	0.1	0.1	7.4	0.3	97.7	24.3	100	0.9
Jaccard	97.5	-	38.5	38.5	97.5	1.0	98.3	96.8	100	73.3
Kulcz.	99.8	62.7	-	100	99.8	47.0	99.8	89.9	100	91.2
Mccon	99.8	62.7	100	-	99.8	47.0	99.8	89.9	100	91.2
Minus	97.8	2.5	0.2	0.2	-	0.3	98.1	24.2	100	1.0
Ochiai	99.6	99.0	54.4	54.4	99.6	-	99.7	99.6	100	84.7
Op	50.0	1.6	0.1	0.1	4.4	0.2	-	16.7	100	0.2
Taran.	76.0	3.3	10.3	10.3	76.1	0.4	83.6	-	100	22.9
Wong3	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	-	0.001
Zoltar	99.1	27.6	13.9	13.9	99.0	16.0	99.8	77.4	100	-

Source: Henrique Ribeiro, 2016

For data-flow and budget 100, Ochiai is significantly better ($p - value \leq 5\%$) than six other heuristics (DRT, Jaccard, Minus, OP, Tarantula and Wong3) and Wong3 is worst than all of them also with significance level of 5%.

Other than using the paired Wilcoxon test, we compared the total amount of defects found by each heuristic for each budget. The defect is check as localized when the number of lines need to be inspected is equal to or less than the budget in question.