On The Impact Of Distinct Metrics For Fault Localization In Automated Program Repair

Marek Mazur

Department of Computer and Systems Sciences

Degree project 15 credits
Computer and Systems Sciences
Degree project at the bachelor level
Autumn term 2022
Supervisor: Mateus de Oliveira Oliveira
Swedish title: Påverkan av fellokaliseringsmått på automatisk programreparation

Stockholm University

Abstract

Automatic Program Repair (APR) is a dynamically growing field of computer science that aims to reduce the time and cost of debugging code and improve its efficiency. Fault localization (FL) is a critical component of the APR workflow and has a real impact on the success of an APR procedure. The fault localization step produces a list of potentially faulty code by calculating its level of suspiciousness, i.e., the likelihood of being faulty. In the process of calculation, a great variety of metrics can be implemented. In this thesis, we examine the effectiveness of ASTOR, a Java APR framework, with chosen FL metrics for calculating suspiciousness by conducting a controlled experiment. ASTOR is tested against the Defects4J dataset, a benchmark for APR evaluation, containing bugs from open-source projects.

Keywords: automated program repair, fault localization, metrics for fault localization.

Synopsis

Background

Automated program repair (APR) is an emerging suite of technologies for automatically fixing errors or vulnerabilities in software systems attempting to improve modern software development by reducing the time and costs associated with program debugging tasks. As the success of a repair may depend not only on the fix itself but also on the location of the fix, fault localization is a pivotal initial step in the APR workflow. A restrictive FL can result in a miss of potential repairs and on the other hand, FL conducted in a permissive manner can cause additional computation. Studies conducted by have shown that inaccurate fault localization happens relatively often in test-based APR and has a real impact on the successfulness of APR tools.

Problem

In this thesis, we take a further look at the APR tool ASTOR, a Java framework that provides explicit extension points to explore the design space of program repair. The fault localization step in ASTOR is delegated to FLACOCO. The calculation of suspiciousness (value to determine how faulty a statement is) involves a metric, i.e., a mathematical formula converting spectra values into a suspiciousness score. To compute the suspiciousness value, FLACOCO involves currently the Ochiai formula exclusively, although a wide range of metrics have been considered and discussed in the literature.

Research Question

How does the use of distinct metrics for fault localization impact the performance of ASTOR?

Method

To address the research question, an experiment involving the evaluation of ASTOR on selected bugs from the Defects4J dataset with six different FL metrics was performed. The tests were executed on an Apple M1 8-core CPU, 16GB

RAM MacBook Pro. The source code and test results are publicly available in the following GitHub repository: https://github.com/mafaust/aprthesis.

Result and discussion

Our experiment showed that the execution time of ASTOR until the first plausible patch was found, could variate based on which metric was used for the fault localization step. The experiment showed also that the mean execution time for the cases when a plausible patch was found could differ from metric to metric. In this context, Barinel resulted as the metric for which the executions of ASTOR were fastest. Based on the results, we could establish that the mean reliability score for the chosen metrics was 58,73 %, and the best score of 66,67 % of reliability was obtained by Ochiai. As it comes to the aspect of suspiciousness correctness, the best score was obtained by Barinel (1007,41 points).

Acknowledgement

I would like to express my gratitude to my supervisor, Mateus de Oliveira Oliveira, for all the guidance, support, and instruction he provided me throughout this project.

Contents

List of Figures

List of Tables

List of Abbreviations

API	–	Application Programming Interface
APR	–	Automated Program Repair
AST	–	Abstract Syntax Tree
CLI	–	Command Line Interface
ESHS	–	Executable Statement Hit Spectra
FL	–	Fault Localization
ML	–	Machine Learning
SBFL	–	Spectrum Based Fault Localization

Chapter 1

Introduction

Debugging code is a resource and time-consuming activity. According to the study conducted by Britton et al. [7], debugging activities often account for nearly 50% of the overall development cost of software products. As reported by NIST - the US National Institute of Standards and Technology - software development costs, including the development and distribution of software patches fixing software bugs, but also costs from lost productivity, contribute to 60 billion US$ economic loss each year in the US alone. Nevertheless, non-material losses have to be taken into account as well. As pointed out by Monperrus [21] software bugs can have fatal consequences and, in a worst-case scenario, cost human lives.

Automated program repair (APR) is an emerging suite of technologies for automatically fixing errors (*bugs*) or vulnerabilities in software systems [14] attempting to improve modern software development by reducing the time and costs associated with program debugging tasks [17]. APR consists of automatically finding a solution to software bugs and generating bug-fixing patches. An automatically identified patch can then be applied with little, or possibly even without, human intervention [21]. APR is a growing and dynamically evolving field of computer science [14] and over the past decade, a wide diversity of automated tooling, analysis and bots have been developed [19, 29, 38, 13, 14, 22]. As identified by Goues et al. [13], the use cases of APR are numerous and can be categorized into four main classes. The first one addresses the problem of fixing bugs throughout development, f.ex. in continuous integration pipelines. In this context, APR could suggest patches in response to regression test failures. The second area is the repair of security vulnerabilities. Memory corruption errors and programming errors are major sources of security flaws, hence a target for automated program repair procedures. Supporting intelligent tutoring systems can be seen as the third area where APR technology could be applied. Within this framework, APR could provide hints to learners for solving programming assignments and automate the process of grading them. Finally, the last area identified by Goues et al. [13] is the integration of APR into self-healing of per-

formance bottlenecks, i.e., online program repair for IoT (Internet of Things) software, as f.ex. support of the repair of non-functional proprieties like energy consumption.

1.1 APR - definitions and key concepts

As stated by Monperrus [21] automated program repair is about *bugs*, which are "deviations between the expected behavior of a program execution and what it actually happened". This umbrella term covers synonymous denominations that can be encountered in the APR literature, i.e., *error*, *failure*, *fault*, *defect* etc. The definition of a bug implies the existence of a deeming entity - a specification, which is a set of expected behaviors. A specification, in turn, is an integral part of the APR workflow and therefore the definition of APR is as follows: "automatic repair is the transformation of an unacceptable behavior of a program execution into an acceptable one according to a specification"[21].

Behavioral and state repair

APR can be categorized into two separate classes: behavioral and state repair[1]. Behavioral repair (BR) consists of changing the behavior of the program under repair, which implies the alteration of the program code. BR can be performed both online or offline under runtime. Commonly, offline BR is performed in the integrated development environment (IDE) or in the context of continuous integration. Online BR, on the other hand, is done on deployed software. BR is strictly related to the use of repair operators and consequently repair models. A repair operator is a type of modification on the program code and a set of repair operators is a repair model. One of the major fields of BR is the test suite-based approach. According to Monperrus [21], in modern object-oriented software, the input can be as complex as a set of interrelated objects built with a rich sequence of method calls, and the output can also be a sequence of method calls that observe the execution state and state and behavior in various ways. In test-suite-based repair, the failing test case acts as a bug oracle, and the remaining passing test cases act as a regression oracle. State repair (SR) consists of altering the execution state of the program, and as opposed to behavioral repair, it is necessarily done at runtime. As an example given by Monperrus [21] "changing the input, the heap, the stack or the environment" can be considered as alterations of the execution. In the context of SR, one of the basic repair concepts is reinitialization and restart, i.e., rebooting a software application in case of failure. Another example is the checkpoint and rollback mechanism, where a rollback (being the repair) restores the program execution to a given checkpoint.

[1] It is worth mentioning that other denominations of this dichotomy exist. One of them is proposed by Gazzola et al. [12] in which *software repair* is opposed to *software healing*.

1.1.1 The workflow of APR

On a superficial level, an APR procedure consists of three phases: fault localization analysis, the generation of plausible patches, and the patch validation phase. An APR tool takes as input a buggy program and a specification, commonly a test suite (but other approaches may be adopted), and, if a fix is generated and validated, produces as output a patch that satisfies the given specification. Otherwise, the user is informed that no solution was found [13].

The APR workflow begins with the fault localization phase, which aims to localize the bug in question and generate a set of possibly faulty program locations, i.e., generate the suspiciousness value for a given program elements (f.ex. statements). As pointed out by Goues et al. [14], virtually all APR techniques begin by narrowing down the program source to a typically imperfect set of buggy locations. Beyond this initial step, APR techniques vary substantially in how they modify the source code in order to address a bug. According to the state-of-the-art, repair techniques concerning phases 2 and 3 of the APR workflow, can be split into generate-and-validate and semantic-driven approaches [12]. The former aims to produce fixes by defining and exploring a space of potential solutions to a given problem. The latter encodes the problem formally and constructs a repair constraint that the patched code should satisfy (f.ex. by conducting a symbolic execution).

Generate-and-validate approaches

A generate-and-validate approach (Figure 1.1) consists of two fundamental steps: the *generate* phase, producing candidate solutions to a given bug, and the *validate* phase, which checks the plausibility of the generated solution. During the *generate* phase, a range of repair operators is used to modify the initial program. Subsequently, a number of new altered program versions are added to the set of candidate solutions. The modifications to the initial program are performed according to the result of the fault localization step, i.e., primarily in the locations identified as buggy. According to the classification by Gazzola et al. [12], the use of repair operators may imply: an atomic change, i.e., a modification of a single element of the code; a code change based on predefined templates or a code alteration using example-based templates, i.e., templates are extracted from historical data. During the *validate* phase, the correctness of the candidate solutions is checked by validating the plausibility of the solution with a given test suite. A program passing all the test cases (including those that initially failed and exposed the bug) is regarded as repaired. The generate-and-validate approach might be executed according to either a *search-based* or a *brute-force* strategy [12]. Applying a search-based strategy results in applying repair operators randomly or with regards to a heuristic algorithm. On the other hand, a *brute-force* strategy results in producing all possible changes that can be obtained by applying a given repair operator.

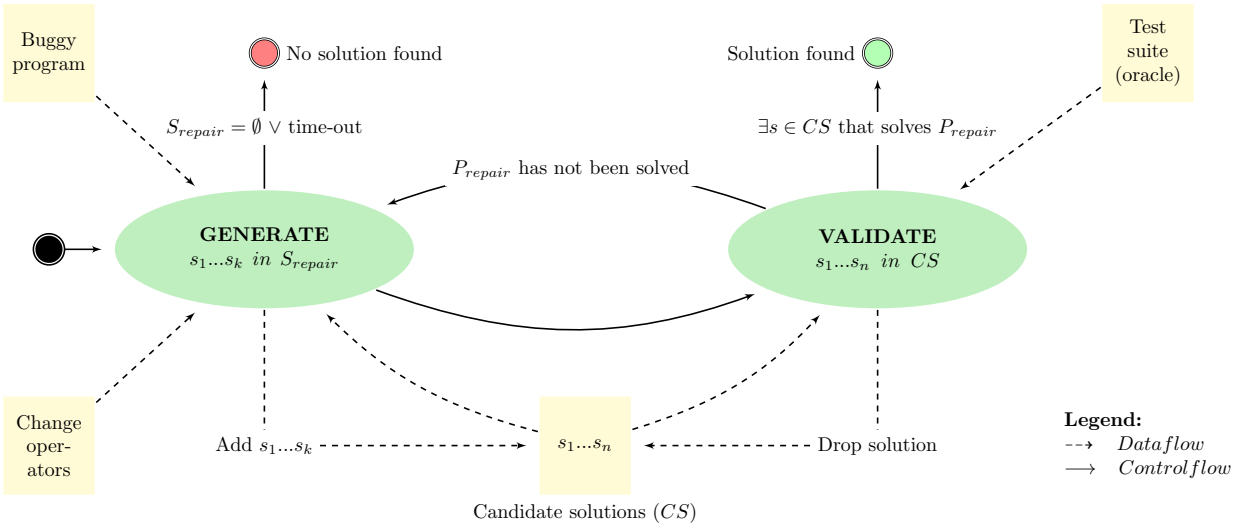

Figure 1.1: Generate and validate workflow as described in [12].

Semantics-driven approaches

To address the problem of software repair, semantics-driven approaches focus on using semantic analysis to construct program patches by applying constraint solving and program synthesis [14]. These techniques encode a problem formally, and when a solution can be found, it is guaranteed to solve a given repair problem, and therefore does not need further validation against an oracle. The workflow of a semantics-driven APR procedure is represented by Figure 1.2 and consists of three essential phases: behavioral analysis, problem generation and fix generation. The behavioral analysis step consists of extracting information about the correct and faulty behaviors from the program under repair. To this end, available test cases (or other forms of specification) and source code of the program under analysis are exploited. Subsequently, the problem generation step involves the gathered data to generate a formal representation of the repair problem. This operation is done either explicitly, i.e., by producing a formula whose solutions correspond to the possible fixes for the given bug, or implicitly, i.e., by conducting an analytical procedure whose outcome is a fix [12]. Finally, the fix generation step attempts to solve the problem generated by the previous phase. As a result, a repairing code change is identified, or the user is informed that a solution is not nonexistent or has not been found (f.ex. due to time restrictions).

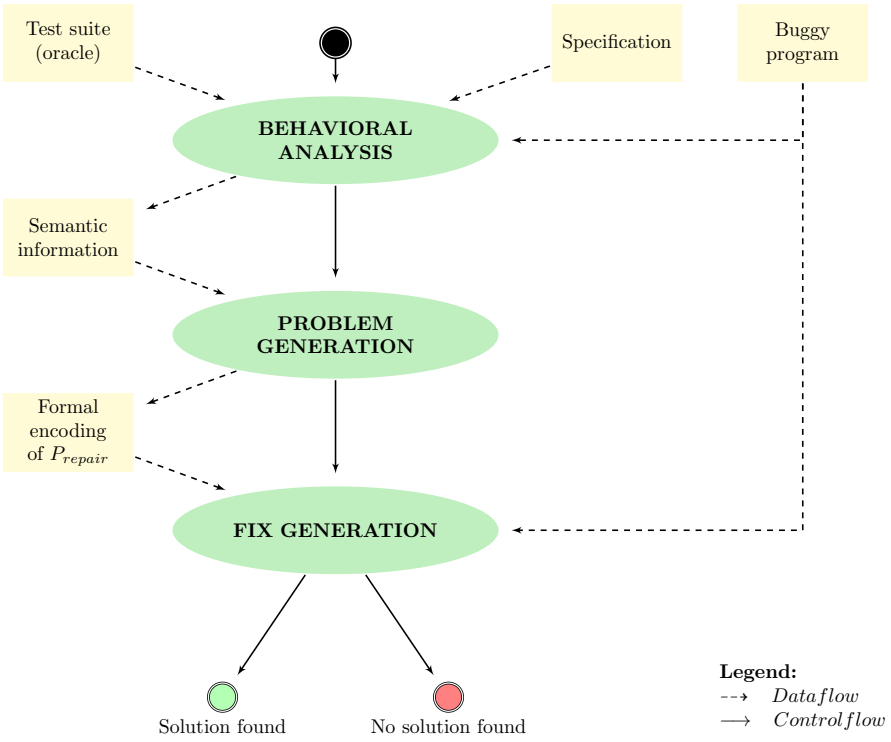

Figure 1.2: Semantic repair workflow as described in [12].

1.2 Research problem

In this thesis, we take a further look at the APR tool ASTOR, developed as a collaboration between KTH Royal Institute of Technology, Inria, the University of Lille, and the Université Polytechnique Hauts-de-France. ASTOR is a Java framework that focuses on the design space of generate-and-validate repair approaches and provides explicit extension points to explore the design space of program repair [19]. The modularity of ASTOR allows us to study how variations of different components affect the performance of the tool. As the success of a repair may depend not only on the fix itself (patch) but also on the location of the fix, fault localization (FL) is a pivotal initial step in the APR workflow. A restrictive FL can result in a miss of potential repairs and on the other hand, FL conducted in a permissive manner can cause additional computation [25]. Studies conducted by Liu et al. [17] have shown that inaccurate fault localization happens relatively often in test-based APR and has a real impact on the successfulness of APR tools. The fault localization step in ASTOR is delegated to FLACOCO, an FL tool built on top of one of the most used and reliable coverage libraries for Java, JACOCO. As FLACOCO imple-

ments the spectrum-based fault localization approach, it produces as output a suspiciousness ranking, indicating how faulty a considered statement might be. The calculation of suspiciousness involves a metric (alternatively called formula), i.e., a mathematical formula converting spectra values (categories of failed and passed test cases) into a suspiciousness score. To compute the suspiciousness value, FLACOCO involves currently the Ochiai formula [2] exclusively, although a wide range of metrics have been considered and discussed in the literature [35, 24]. Comparison with alternative metrics and further investigation of this area are needed.

Research question. *How does the use of distinct metrics for fault localization impact the performance of ASTOR?*

To address the formulated research question, an experiment involving the evaluation of ASTOR on a chosen bug dataset with fault localization metrics will be performed. In order to conduct the experiment, modifications to the FL tool FLACOCO will be made and new metrics will be added. As ASTOR and FLACOCO are well-integrated no further changes are needed on the side of the APR framework. The impact of the chosen metrics on the effectiveness of ASTOR will be measured in terms of time of execution, reliability of the metric and suspiciousness correctness.

We believe that our experimental results may bring valuable insights regarding the influence of fault localization metrics on the effectiveness of APR tools, such as ASTOR. We also expect that our experiments may help to elucidate the role of fault localization and suspiciousness metrics in the APR workflow.

1.3 Structure of the thesis

The remainder of this thesis is organized as follows. In order to situate the forthcoming research within the existing theoretical framework, in chapter 2, we highlight the relevant literature regarding fault localization in software repair and the diversity of metrics. In chapter 3, we provide a review of the key features and the workflow of ASTOR, i.e., the APR tool in which the experiment will be conducted. Subsequently, in chapter 4, we outline the adapted experimental framework. Finally, in chapter 5, we present the results of the experiments, and in chapter 6, we discuss the research findings, followed by final conclusions and avenues for future research.

Chapter 2

Fault localization

According to the extensive survey by Wong et al. [35], software FL techniques can be divided into traditional and advanced FL. The former category relates to intuitive FL tools for bug identification, such as program logging, assertions, breakpoints, and profiling. The latter refers to techniques implemented, among others, in the APR workflow. These techniques are categorized into eight subcategories: spectrum-based FL, slice-based FL, statistics-based FL, program state-based FL, machine learning-based FL, data mining-based FL, model-based FL, and additional FL techniques which focus on specific program languages or testing scenarios. The following section will focus on spectrum-based fault localization as the experimental part of the thesis employs this FL approach.

2.1 Spectrum-based fault localization

Spectrum-based[1] fault localization (SBFL) is an FL technique widely used in APR tools and considered one of the best-studied ones [35, 12]. In this approach, information about the software's behavior is collected by testing it and evaluating its likeliness to be faulty by calculating a suspiciousness value. By involving techniques such as code coverage or ESHS (executable statement hit spectra) that indicate which elements have been covered during a program execution, SBFL techniques can significantly narrow the search space of faulty components. In SBFL, program elements (such as program statements) executed by many failing tests and few passing tests are considered potentially faulty. On the contrary, program elements executed by few failing tests and many passing tests are considered potentially correct. In order to calculate the suspiciousness value of a given statement, various metrics have been proposed over the years [35].

[1] A program spectrum is a collection of data that provides a specific view on the dynamic behavior of software[3].

2.2 SBFL metrics

Modern SBFL metrics involve both failing[2] and passing test cases, as the exclusive use of failing test cases in the past has been proven to be inefficient [32, 35]. Using techniques such as set intersections and set unions, the relationship between the two test cases groups can be manipulated [35]. As a result, the following spectrum variables are utilized to calculate the degree of suspiciousness $S(s)$ of a given statement s :

Variable	**Explanation**
$total\ passing$	Total number of passing test cases (in the test suite)
$total\ failing$	Total number of failing test cases (in the test suite)
$passing(s)$	Number of passing test cases that the statement s covers
$failing(s)$	Number of failing test cases that the statement s covers

Table 2.1: Spectrum variables.

2.2.1 Computation of suspiciousness

A test suite with passing and failing test cases is a prerequisite for the SBFL approach, where failing tests reveal the fault that must be localized and repaired [12] and passing test cases represent correct behaviour. The role of the SBFL procedure is to return a suspiciousness score for each considered statement, i.e., a value between 0 and 1[3], where 0 represents the least suspicious statement, and 1 is the most suspicious one. In this study six metrics are presented, as they are considered as well-studied and widely implemented in the FL step, being part of the APR workflow [24, 35, 12]. The following table presents the chosen metrics and their algebraic form.

[2]The execution of a failing test case exposes the bug.

[3]In some cases, as f.ex. with the D* and Op2 metrics, the suspiciousness score might exceed these boundaries.

Metric	**Formula**
Ochiai[2]	$S(s) = \frac{failing(s)}{\sqrt{total\ failing \cdot (failing(s)\ +\ passing(s))}}$
Barinel[4]	$S(s) = 1 - \frac{passing(s)}{passing(s) + failing(s)}$
D*[33]	$S(s) = \frac{failing(s)^*}{passing(s) + (total\ failing - failing(s))}$
Op2[23]	$S(s) = failing(s) - \frac{passing(s)}{passing(s) + (total\ passing + 1}$
Jaccard[12]	$S(s) = \frac{failing(s)}{(failing(s) + passing(s)) + (total\ failing - failing(s))}$
Tarantula[15]	$S(s) = \frac{failing(s)/total\ failing}{failing(s)/total\ failing + passing(s)/total\ passing}$

Table 2.2: FL metrics and their algebraic form.

To illustrate the process of evaluating suspiciousness with the chosen metrics, an example (Table 2.3) considering a buggy Java method is presented. The method takes as input two integers (x, y) and returns four possible outputs: *Positive*, if the sum is a positive integer; *Zero*, if the sum equals to 0; *Negative*, if the sum is a negative integer; *Invalid input*, if any of the two integers passed as parameters is greater than 10. Twelve test cases are involved, eight qualified as passing and four qualified as failing (used input values for given test cases are available below, Table 2.4). The symbol $\circ$ represents the execution of a given statement for passing test cases. To emphasize the execution of the failing test cases the symbol is $\bullet$ instead.

s_{id}	Buggy Java method	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	Och.	Bar.	D*	Op2	Jac.	Tar.
s_1	*public void sum(int x, int y){*	•	•	•	•	○	○	○	○	○	○	○	○	0.57	0.33	2.00	3.11	0.33	0.50
s_2	*if(x < 10 && y < 10){*	•	•	•	•	○	○	○	○	○	○	○	○	0.57	0.33	2.00	3.11	0.33	0.50
s_3	*int result = x − y; // buggy*	•	•	•	•		○	○	○	○	○	○	○	0.60	0.36	2.28	3.22	0.36	0.53
	// instead: x + y																		
s_4	*if(result > 0){*	•	•	•	•		○	○	○	○	○	○	○	0.60	0.36	2.28	3.22	0.36	0.53
s_5	*System.out.println("Positive");*	•					○	○			○	○	○	0.20	0.16	0.13	0.44	0.11	0.29
s_6	*} else if (result == 0) {*		•	•	•				○	○				0.67	0.60	3.00	2.78	0.50	0.75
s_7	*System.out.println("Zero");*		•							○				0.35	0.50	0.25	0.88	0.20	0.66
s_8	*} else { System.out.println("Negative");}*			•	•				○					0.58	0.66	1.33	1.89	0.40	0.80
s_9	*} else { System.out.println("Invalid input");)}*					○								0.00	0.00	0.00	0.00	0.00	0.00
	}																		

Table 2.3: Example of suspiciousness computation.

Test case	Input	Expected output	Actual output	Qualification
t_1	$x = 4, y = 5$	*Negative*	*Positive*	failing
t_2	$x = -1, y = -1$	*Negative*	*Zero*	failing
t_3	$x = 4, y = 5$	*Positive*	*Negative*	failing
t_4	$x = 3, y = 4$	*Positive*	*Negative*	failing
t_5	$x = 11, y = 11$	*Invalid input*	*Invalid input*	passing
t_6	$x = 5, y = 1$	*Positive*	*Positive*	passing
t_7	$x = 4, y = 2$	*Positive*	*Positive*	passing
t_8	$x = -2, y = 1$	*Negative*	*Negative*	passing
t_9	$x = 0, y = 0$	*Zero*	*Zero*	passing
t_{10}	$x = 6, y = 3$	*Positive*	*Positive*	passing
t_{11}	$x = 6, y = 2$	*Positive*	*Positive*	passing
t_{12}	$x = 6, y = 1$	*Positive*	*Positive*	passing

Table 2.4: Input for test cases.

2.3 Beyond spectrum-based FL techniques

Slice-based FL involves program abstraction that aims to reduce the program size and thus, in the context of FL, reduce the search space for buggy elements [35]. First publications about slicing date back to 1979, when M. Weiser proposed the static variant of this technique [30]. The intuition behind it is that if a test case fails due to, f. ex., an incorrect variable value at a given statement, the bug should be found in the static slice associated with that statement. The drawback of static slicing is that a slice will always contain all the executable statements that are related to and can influence a given variable. To address this issue, dynamic slicing involves the computation of slices for given input or executions [4]. To illustrate the process of slicing and the difference between the two variants, an example with a buggy Java method is presented in Table 2.5.

[4]Execution slicing can be a category on its own

s_{id}	Buggy Java code snippet
s_1	$int\ x\ =\ scanner.nextInt();$
s_2	$int\ i\ =\ 1;$
s_3	$int\ sum\ =\ 0;$
s_4	$int\ product\ =\ 1;$
s_5	$while(i\ <=\ x)\{$
s_6	$sum\ =\ sum\ +\ i;$
s_7	$product\ *=\ 2 * i;\ //\ buggy$
	$//instead\ product\ *=\ i;$
s_8	$i\ +=\ 1;\ \}$
s_9	$System.out.println("sum");$
s_{10}	$System.out.println("product");\ \}$
	Static slicing for variable *product*
s_1	$int\ x\ =\ scanner.nextInt();$
s_2	$int\ i\ =\ 1;$
s_3	
s_4	$int\ product\ =\ 1;$
s_5	$while(i\ <=\ x)\{$
s_6	
s_7	$product\ *=\ 2 * i;\ //\ buggy$
s_8	$i\ +=\ 1;\ \}$
s_9	
s_{10}	$System.out.println("product");\ \}$
	Dynamic slicing for variable *sum* and input $x = 0$
s_1	$int\ x\ =\ scanner.nextInt();$
s_2	$int\ i\ =\ 1;$
s_3	$int\ sum\ =\ 0;$
s_4	
s_5	$while(i\ <=\ x)\{\}$
s_6	
s_7	
s_8	
s_9	$System.out.println("sum");$
s_{10}	

Table 2.5: Example of static and dynamic slicing.

Variables representing the state of a program at a given execution point can also be a valuable resource for fault localization [35]. In **state-based FL**, relative debugging [1] is considered one of the main techniques. It involves a runtime comparison of internal states between a development version and a *reference* version of a chosen program. Alternatively, the modification of variables' values can be involved, by which it is possible to determine the cause(s) of erroneous executions [42, 41].

Being adaptive and robust, machine learning (ML) techniques aim at producing data-based models [35]. These characteristics led to extensive use of ML in research and industry. The problem of fault localization with the application of ML can be expressed as the problem of deducing the location of a bug by analyzing input data such as execution results (spectrum variables) and code coverage [28, 36, 34].

Data mining, a technique frequently combined with machine learning, involves extracting information from large data sets and can help effectively reveal patterns in data [35]. In order to pinpoint the localization of a fault, it's often necessary to study the entire execution path of an application. Doing this through practical means is difficult due to the volume of data involved. Instead, researchers have turned to data mining methods to study statement execution. **Data mining-based FL** approaches are helpful as the pattern leading to software crashes can be isolated [40, 9, 8].

Model-based FL techniques rely on available models of the programs being diagnosed [35]. These models determine a program's functionality, and consequently they act as oracles for the programs. Each model-based technique has to account for the expressive capabilities of its chosen model. This consideration influences how effective each technique is in diagnosis. To discover bugs in a program, models are created based on the actual software being modeled. These models use differences between the program's behavior and a model's expected behavior to find flaws in the software. Conversely, models created directly from the actual software may contain bugs — making them useful for fault localization. By looking at differences between the program's actual behaviors and how it was intended to behave, programmers or testers can identify which model elements cause errors in the software [10, 20, 37].

Chapter 3

ASTOR

3.1 The design of ASTOR

ASTOR is a framework that allows researchers to implement new automated program repair approaches, and to extend available ones [19]. Being a validate-and-generate tool, ASTOR first performs a search within a search space to generate a set of patches, and subsequently validates them using a correctness oracle. An approach over ASTOR requires as input a buggy program to be repaired and a correctness oracle such as a test suite. As output, the approach generates, when it is possible, one or more patches that are valid according to the correctness oracle. As the thesis is focused on the fault localization step of the APR workflow, we discuss in more detail the structure of FL tool FLACOCO (Figure 3.2).

Superficial workflow and extension points

Figure 3.1 illustrates the workflow of ASTOR on a superficial level. From the buggy program, an abstract syntax tree (in the context of ASTOR, it is called a spoon model) is created. The next step of the execution is the fault localization step, and the computation of suspiciousness values of program elements considered buggy. Consecutively, program variants are created, and modification points are assigned to the program elements identified as buggy by the FL step. At this point, ASTOR begins the generate-and-validate procedure (represented as a while loop in Figure 3.1). The generate phase consists of selecting a modification point and a repair operator, and applying the chosen repair operator on the modification point to generate a candidate patch. Subsequently, begins the validation phase. It consists of integrating the generated patch with the input program and validating it against the correctness oracle (test suite). As the probability of finding a plausible patch with the first generated variant is relatively low, the generate-and-validate phase is repeated until either a plausible patch is found or a time-out value is reached. If multiple patches have been found, then as a final step, they are sorted in chronological order (as default).

For each of the aforementioned steps, ASTOR includes extension points, i.e., easily accessible APIs, usually in the form of Java interfaces to implement or abstract classes to extend, that permit to modify the APR process. The extension points give access to the fault localization strategy; the granularity of modification points; the navigation strategy; the selection of suspicious modification points; the selection of repair operator; the ingredient pool definition; the selection of ingredients; the ingredient transformation; the candidate patch validation phase; the patch post-processing. In order to execute existing or newly created extension points, the name specifying it has to be passed as a command line argument.

Algorithm 1 Main steps of *generate-and-validate* repair approaches, implemented in Astor (extension points are referred to as comment prefixed by `//`,)

Require: Program under repair P
Require: test suite TS
Ensure: A list of test-suite adequate patches

```
1: suspicious ← run-fault-localization(P, TS) //EP_FL
2: mpl ← create-modification-points(suspicious) //EP_MPG
3: ops ← get-operators() //EP_OD
4: tsa-patches-refined ← ∅
5: nr-iteration ← 0
6: starting-time ← System.currentTime
7: while continue-searching(starting-time, nr-iteration, size(tsa-patches)) do
8:    program-variants ← generate-program-variants(P, mpl, ops)
9:    tsa-patches ← tsa-patches + validate-variants(P, program-variants, TS)
10:   nr-iteration ← nr-iteration + 1
11: end while//EP_NS
12: tsa-patches-refined ← refining-patches(tsa-patches) //EP_SP
13: return tsa-patches-refined
```

Figure 3.1: The high-end workflow of ASTOR [19].

3.1.1 Fault localization

The workflow of ASTOR begins with the fault localization step, which is based on spectrum analysis. The default FL tool for ASTOR is FLACOCO, built on top of one of the most used and most reliable coverage libraries for Java, JACOCO [26]. FLACOCO can be accessed either through a command line interface or an API written in Java (as is the case of ASTOR) that integrates with other applications written in languages compiled to Java bytecode (phase 1) [26]. In the test detection phase (phase 2), FLACOCO parses the compiled classes of a given project in search of test cases. It filters out test cases by scanning the complied classes according to the test framework's specifications.

FLACOCO can detect tests written with JUnit 3, JUnit4, and JUnit5, and according to the authors, it is the only FL tool with such wide test driver support [26]. Technically, test discovery is achieved by starting a new Java thread with a custom class loader and setting up a classpath with only the inspected project's classes and dependencies.

In the instrumentation and test execution phase (phase 3a and 3b), FLACOCO executes as a separate process all the previously detected tests. The notion of instrumentation (of the execution) relates to the fact that during this phase, code coverage data is collected for each test case, i.e., data is gathered to be utilized by FLACOCO at a later stage. In FLACOCO, this procedure is done using a custom Java agent (a specialized type of class that intercepts applications running on the Java Virtual Machine). Instrumentation of the executed classes takes place at class loading time during test execution and is fully provided by the Java code coverage library JACOCO. As the final part of this stage (phase 3c), FLACOCO collects for each unit test: the result of the test, whether it passed or failed; data about exceptions, if any was thrown; code coverage data (at a statement granularity) for all executed classes by the given test. In the subsequent phase (phase 4), the obtained test data is loaded for further processing, i.e., the computation of the suspiciousness of each line of code. According to Silva et al. [26] the score of a line is based on the tests that covered it, the results of those tests, and a specified formula. For this purpose, FLACOCO uses as default the Ochiai metric. In conclusion (phase 5), FLACOCO produces an output. Based on the access point (API/CLI), the output has the form of either an AST annotated with suspiciousness information or exportable files in formats such as CSV or JSON.

3.1.2 Identification of modification points

Once the fault localization step returns a list of suspicious code locations, ASTOR creates a representation of the program under repair. During this phase modification points are created from the statements returned from the FL step and regarded as suspicious (with a high suspiciousness level). Different generate-and-validate approaches will create different modification points.

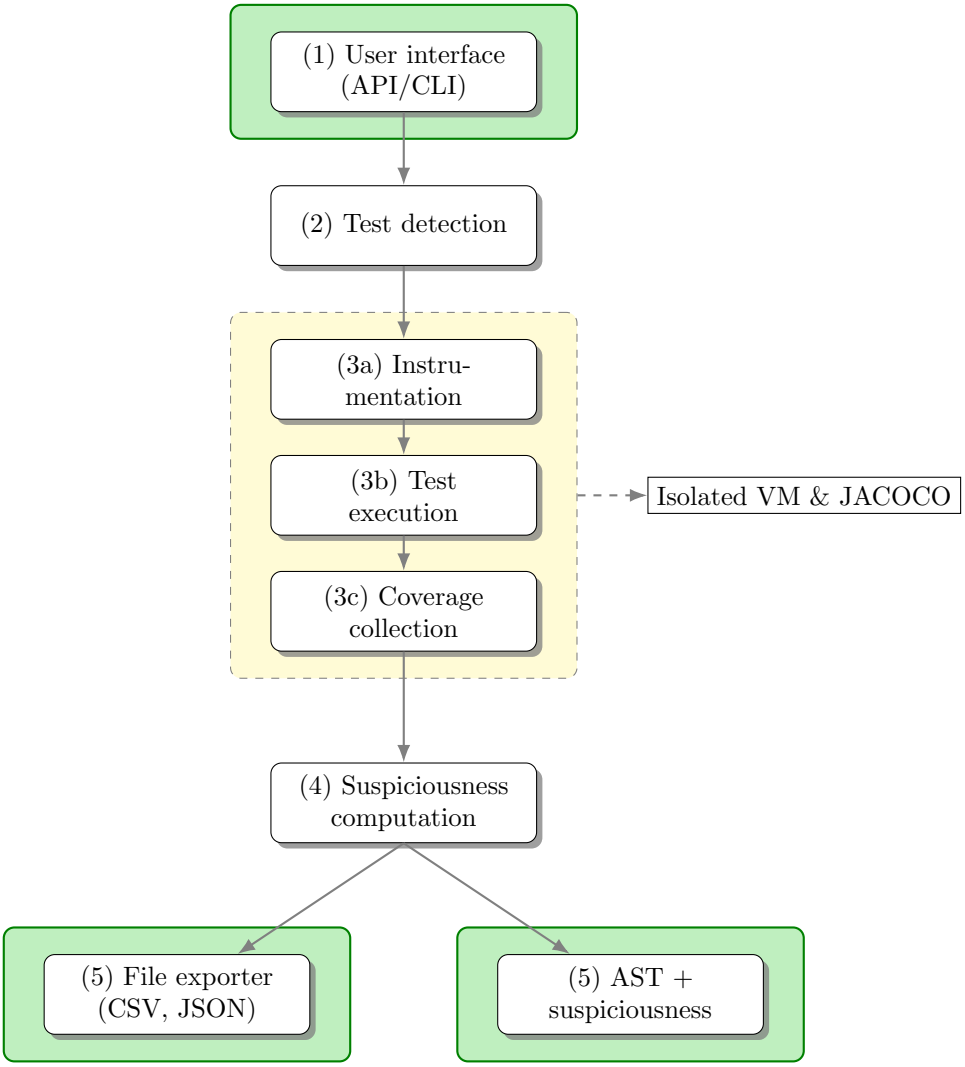

Figure 3.2: The architecture of FLACOCO.

3.1.3 Creation of repair operators

This phase begins by applying automated code transformation over modification points. These transformations are done by repair operators[1] and the set of repair operators constitute the repair model. The application of repair operators depends on the chosen generate-and-validate approach. For instance, jMutRepair will alter a logical operator without the use of any additional infor-

[1] A repair operator is an action that transforms a code element associated to a modification point into another [19], f.ex. the substitution of a logical operator > to >=.

mation. On the other hand, such as in the case of jGenProg and DeepRepair, some extra information (f.ex. code snippets coming from other location in the source code), commonly known as *ingredient*, might be needed before a code transformation can be applied.

3.1.4 Navigation of the search space

Once that that all code transformations to be applied to a suspicious element are known, ASTOR proceeds with the navigation of the search space [19]. The goal is to find, between all possible modified versions of the buggy program, one or more versions that do not contain the bug under repair and that do not introduce new bugs.

3.1.5 Generation of program variants

The creation of program variants in ASTOR consists of the following steps [19]:

1. Selection of modification points: according to a selection strategy, at least one modification point is chosen where repair operators can be applied.

2. Selection of repair operators: for each selected modification point, one repair operator is applied at the specified location. Consequently, the created transformation is added to the set of code transformations.

3. Creation of code transformation[2]: as a code transformation might need ingredients, ASTOR begins this phase by detecting if a repair operator needs one or not. In the former case, the program execution continues, and ASTOR returns the transformation composed by the modification point and the repair operator. In the latter, ASTOR creates a so-called *ingredient pool* consisting of elements of the program under repair. The procedure of collecting ingredients consists of parsing the code (at a given granularity) and storing each parsed element in the pool. The next step consists of querying the ingredient pool by the repair approach when an ingredient-based repair operator needs an ingredient for synthesizing the candidate patch code. Moreover, when an operator gets an ingredient from the ingredient pool, it can use it directly or after applying a transformation over the ingredient.

4. Creation of program variants: program variants are generated from the aforementioned code transformation phase.

3.1.6 Candidate patch validation

Following the creation of program variants, the candidate patch validation begins. This phase aims to return test-suite adequate patches. To do so, ASTOR

[2]A code transformation is a concept that groups a modification point *mp* and a repair operator *op* [19].

synthesizes the code source of the patch from each variant and applies it to the buggy version of the program under repair [19]. Lastly, it evaluates the altered version with the correctness oracle. As a result, the patch is classified as a fix and stored (on the condition that it passes the correctness check).

Evaluation of conditions for ending navigation

ASTOR finishes the search for plausible patches given that one of the following conditions, configured by the user or set as default values, is fulfilled [19]:

1. A given number of plausible patches is found.
2. The execution of the program has reached a maximum number of iterations.
3. The execution of the program has reached a time limit (timeout).

3.1.7 Solution post-processing

Based on the jGenProg approach [19], this phase aims primarily to remove unnecessary code alterations and keep only those considered essential to the bug's repair. Furthermore, sorting patches according to a given criterion is also possible. ASTOR, by default, lists patches in chronological order. However, the tool can list patches according to other criteria, such as the number or type of modification.

Chapter 4

Methodology and experimental framework

In this chapter aspects concerning the experimental framework, i.e., research strategy, data collection and the analysis method are presented. To address the research question formulated in section 1.2, an experiment involving the evaluation of ASTOR on a chosen bug dataset with six different FL metrics was performed. The experiment examined if a given fault localization metric has an impact on the effectiveness of the APR tool ASTOR and was conducted in accordance with the empirical methodology described in *Experimentation in software engineering* [31]. The tests were executed on an Apple M1 8-core CPU, 16GB RAM MacBook Pro. The experiment was prepared in the IntelliJ integrated development environment and run via the command line interface (CLI). The source code and test results are publicly available in the following GitHub repository[1]: https://github.com/mafaust/aprthesis.

4.1 APR and FL software

The experiment involved the APR framework ASTOR and the fault localization tool FLACOCO, which implements the spectrum-based fault localization approach. At the moment, ASTOR incorporates six generate-and-validate approaches, i.e., jGenProg, jMutRepair, jKali, DeepRepair, Cardumen and 3sfix. The experiment was performed using the jGenProg generate-and-validate approach exclusively. As ASTOR is published under a GNU license and FLACOCO under an MIT license, both programs are open-access and available for commercial and private use.

[1] Link to the software heritage version of the repository: https://archive.softwareheritage.org/browse/origin/directory/?origin_url=https://github.com/mafaust/aprthesis

4.2 Modifications to the software

In order to perform the experiment, several modifications had to be made to the fault localization tool FLACOCO. Firstly, Java classes representing the new metrics had to be written and added to the tool. The newly added metrics (formulas) had to implement the interface *Formula* and had to be added to the Enum Java class *SpectrumFormula*. The Enum values were subsequently used in the FLACOCO configuration class *FlacocoConfig*. The last step in the modification process was adding the various metrics to the configuration class. Due to the fact that FLACOCO and ASTOR are well-integrated, no further alterations had to be made on the side of the APR tool.

4.3 Fault localization metrics

Wong et al. [35] lists over 30 fault localization suspiciousness metrics (similarity coefficient-based techniques). Among them, five metrics are regarded as best-studied and broadly implemented. Those are Ochiai, Barinel, DStar, Op2 and Tarantula. According to Gazzola et al. [12], the Jaccard metric is considered a relatively well-studied metric and has been implemented in various APR tools. As the Ochiai metric is the standard formula implemented in the FLACOCO FL tool, the experiment will involve five additional metrics: Barinel, DStar, Op2, Tarantula and Jaccard.

4.4 Bug dataset and bug collection

The chosen bug dataset is Defects4j [16], i.e., a collection of reproducible bugs embedded in open-source projects, and an experimental infrastructure facilitating controlled experiments in software engineering research. Defects4j has become, during the last decade, a standard benchmark for Java-based APR tools and has been widely used in research [18, 27, 39, 6, 11, 5]. The dataset is divided into the following projects: Chart (number of active bugs: 26), Cli (39), Closure (174), Codec (18), Collections (4), Compress (47), Csv (16), Gson (18), JacksonCOre (26), JacksonDatabind (112), JacksonXml (6), Jsoup (93), JxPath (22), Lang (64), Math (106), Mockito (38), Time (26).

During the collection phase, a selection of bugs had to be performed. The criteria were as follows:

1. The bug is located within the Math project [2].
2. For a given bug, the jGenProg approach has generated a plausible patch in past evaluations.

[2] According to the results published by Martinez and Monperrus [19], ASTOR has produced the major amount of plausible patches for the Math project. Furthermore, Math is one of the biggest collections of bugs in the Defects4J environment.

3. The bug is considered *active* according to the information published by the developers of Defects4j on their GitHub repository, i.e., the code for the given bug is not deprecated.

4. The bug has passed the compile-and-test phase by running commands provided by Defects4j.

4.5 Data collection method

All tests were run with a set of predefined conditions determining the behavior of ASTOR. Once the execution of ASTOR was completed, or the time-out value was reached, data was generated in several files. From the research point of view, three of them are of particular interest: the JSON file (astor_output.json) containing statistical data regarding the execution and the generated patch; the JSON file (suspicious_*bugcode*.json) containing the suspicious code list generated by the fault localization step; the log file containing the terminal output generated during the execution. The first of the listed files is valuable as it contains the execution time of the APR tool (fault localization step excluded) and detailed information about the generate-and-validate step. The data contained in the second file gives an insight into which statements are classified as suspicious by the fault localization step and how the classification of suspiciousness varies based on the chosen metric. The data is organized in the form of a sorted list based on the suspiciousness value (most suspicious statements first). Finally, the log file documents the process of executing ASTOR and gives insight into the workflow of the APR tool. All generated files are stored in the project's GitHub repository.

4.6 Data analysis method

The gathered data was prepared for analysis by extracting the relevant information from the JSON files into CSV files. As this operation was carried out manually, the reliability of the data could be compromised. Further investigation was done within the GraphPad Prism software environment.

Three parameters are proposed to evaluate the impact of the chosen metrics on the effectiveness of ASTOR:

1. Performance, i.e., the time required to find the first plausible patch for a given bug. The performance parameter is expressed in seconds and excludes the execution of the fault localization step.

2. Reliability of the metric, i.e., the number of bugs ASTOR can repair with the given metric compared to the total amount of bugs considered in the experiment. This parameter is expressed as a percentage value.

3. Suspiciousness correctness, i.e., the level of accuracy to assign the buggy statement a high suspiciousness rank in the fault localization step. This parameter is calculated with the following formula:

$$Suspiciousenss\ correctnes = \frac{total\ suspicious}{ranking(s))}$$

where *total suspicious* is the total amount of all statements identified as suspicious and $ranking(s)$ is the position of the buggy statement s in the suspiciousness ranking list. The intuition behind the formula is that the higher the faulty statement is ranked, the bigger the score of suspiciousness correctness is for the given metric. Furthermore, the value is also dependent on the amount of statements identified as suspicious. Two special cases have to be considered when evaluating this parameter. First, if the buggy statement is not identified as suspicious, i.e., it's suspiciousness rank equals 0, then the suspiciousness correctness value should also equal 0. Second, if the buggy statement's suspiciousness rank is among statements evaluated as equally suspicious, then it's position in the suspiciousness ranking list is the first one with the given value.

4.7 Alternative research strategy

According to Wohlin et al. [31], a case study is a plausible alternative for an experiment. This approach is well suited for a variety of problems within software engineering, as the studied phenomena are usually contemporary objects. Although case studies typically produce qualitative data and, therefore, cannot be used to explain causal relationships, they can provide a deeper understanding of the phenomena under study. Because scale-up problems can be avoided, a case study is especially adapted for evaluating software engineering methods and tools in a real-life context within a specific time. The adaptation of this methodology could help determine not only how the fault localization step in an APR tool is performed but also what are the differences between the applications of selected FL metrics and why under given circumstances, they might have an impact on an APR tool. As case studies differ from controlled empirical approaches, the generalizability of the findings might be limited and biased by researchers. The mentioned concerns can, however, be addressed by applying high-quality research practices and thorough documentation of the research process.

Chapter 5

Results

5.1 Performance

The first considered aspect is the performance of ASTOR with the chosen metrics. According to Table 5.1, 126 executions of ASTOR have been performed in total, i.e., 21 executions per metric. 74 executions (58,7 %) found a plausible patch for the buggy code. On the other hand, 52 executions (41,3 %) reached the time-out value and therefore didn't find a solution. The executions of ASTOR with the Barinel metric have resulted as the fastest (122,72 s). This result is followed by Jaccard (175,23 s), Op2 (190,62 s), Ochiai (283,08 s) and Tarantula (351,29 s) metrics. The slowest performance was achieved by running ASTOR with the D* metric (512,81 s).

As presented in Table 5.2, the greatest number of fastest executions (5) was achieved by Barinel and Jaccard, followed by Tarantula, which was the fastest in two cases (Math 40, Math 82). Finally, Ochiai, Op2, and D* had only one fastest execution per metric. Moreover, a collection of scatter plots is presented in Figure 5.1, showing how the performed executions relate to the mean time value. The illustrated results don't include the time-out values and the bugs for which none of the metrics have found a solution. The variance of ASTOR execution time between the metrics is furthermore illustrated with a heatmap (Figure 5.2), where colors represent clusters of time.

Bug Id	Barinel	D*	Jaccard	Ochiai	Op2	Tarantula	Fastest execution
Math 2	210,00	625,64	216,39	224,88	223,82	211,67	Barinel
Math 5	220,82	*3600,04*	220,44	227,04	283,46	285,90	Jaccard
Math 7	*3602,58*	*3602,55*	*3600,35*	371,23	*3600,32*	*3669,84*	Ochiai
Math 8	213,08	513,66	204,65	214,19	221,08	220,20	Jaccard
Math 20	*3740,90*	*3602,33*	*3601,12*	*3601,63*	*3768,50*	*3600,96*	N.A.
Math 32	*3602,76*	*3601,80*	*3603,07*	*3604,19*	*3603,35*	*4253,33*	N.A.
Math 39	*3604,19*	*3600,04*	*3605,56*	*3601,48*	*3604,89*	*3601,85*	N.A.
Math 40	285,27	2757,54	276,77	684,37	*5334,81*	206,24	Tarantula
Math 44	*3602,83*	*3600,40*	*3601,53*	*3600,65*	*3602,42*	*3839,00*	N.A.
Math 64	348,77	*3601,29*	*3600,70*	838,28	*3602,40*	*3601,78*	Barinel
Math 70	35,60	99,02	36,34	50,13	263,62	60,03	Barinel
Math 71	*13863,31*	*3600,21*	474,96	771,78	530,26	2441,09	Jaccard
Math 73	26,99	116,92	26,79	45,55	281,46	261,84	Jaccard
Math 74	*3605,46*	*3602,34*	*3603,11*	*3606,50*	*3607,12*	*3603,13*	N.A.
Math 78	*3600,10*	*3600,30*	*3622,60*	*4123,61*	*3893,04*	*3613,07*	N.A.
Math 80	41,80	134,56	44,10	48,45	49,97	82,05	Barinel
Math 81	50,16	201,63	50,04	153,89	48,72	260,83	Op2
Math 82	55,61	571,31	55,94	213,77	57,86	18,08	Tarantula
Math 84	32,61	547,97	597,26	*3602,56*	*3618,57*	*3601,85*	Barinel
Math 85	22,30	14,09	22,02	17,77	64,13	94,85	D*
Math 95	52,37	58,58	52,35	101,85	72,46	72,65	Jaccard

Table 5.1: Time of execution (s) of ASTOR for the chosen fault localization metrics. The executions that reached the time-out value are marked with the cursive font. The last column presents the name of the metric that had the fastest execution for the given buggy program. N.A. (not applicable) represents the cases when none of the metrics found a plausible patch.

	Barinel	D*	Jaccard	Ochiai	Op2	Tarantula
Mean time of execution	122,72	512,81	175,23	283,08	190,62	351,29
Total fastest executions	5	1	5	1	1	2

Table 5.2: Performance summary.

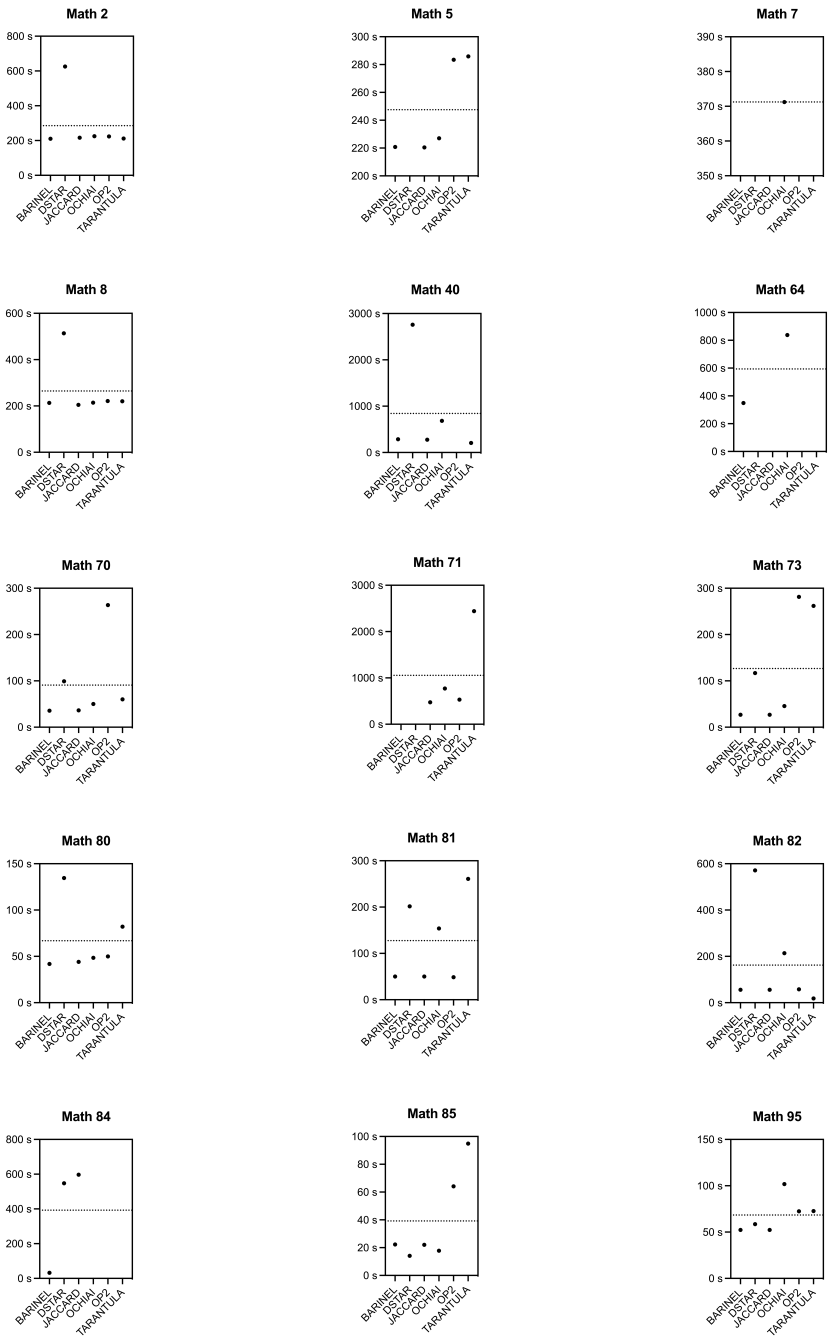

Figure 5.1: Results grouped by test. The dotted line represent the mean execution time.

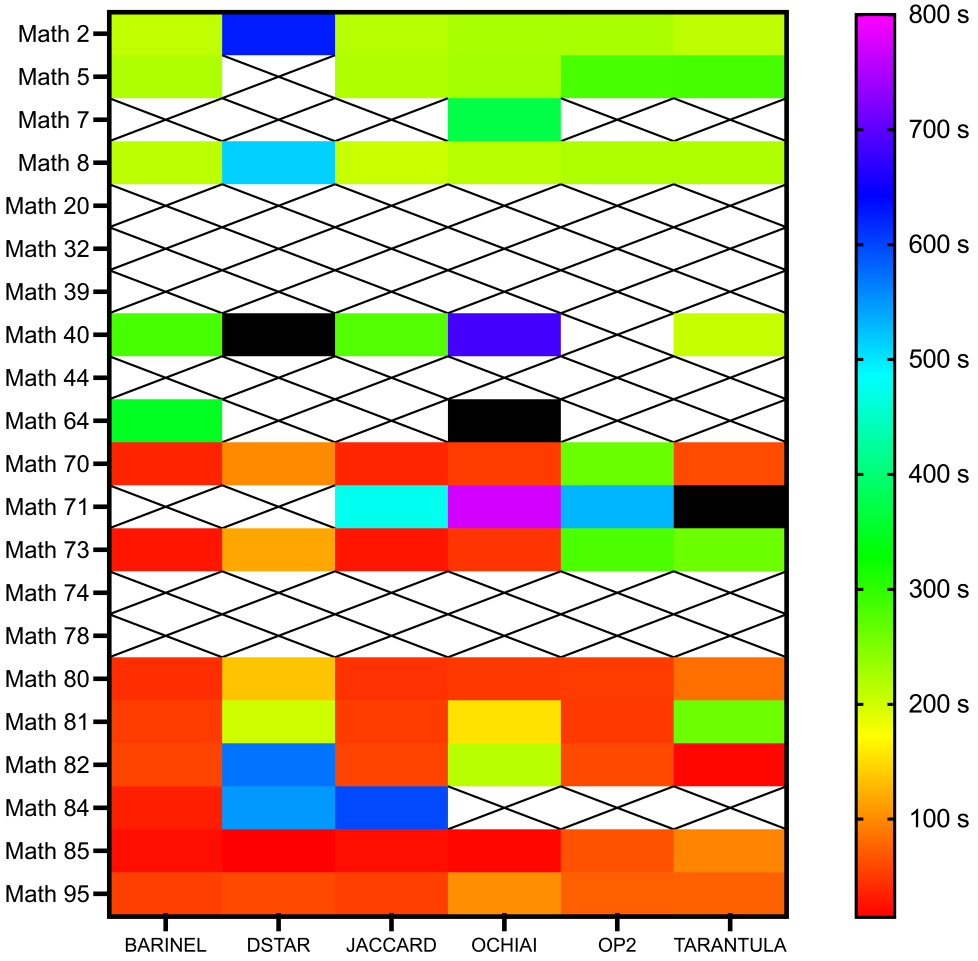

Figure 5.2: Performance heatmap illustrating the variance of ASTOR execution time. X-boxes represent the cases when the time-out value was reached. Black boxes represent all the executions over 800 s.

5.2 Reliability of the metrics

According to the results summarized in Table 5.3, Ochiai, implemented as the default metric in the Flacoco fault localization tool, found the greatest number of patches, i.e., in 14 out of 21 possible cases. Nonetheless, the executions involving the remaining metrics resulted in 13 (Barinel, Jaccard), 12 (Tarantula), and 11 (D*, Op2) found patches. The difference between the best and the worst result is thus of only three plausible patches. The results mentioned above were used to calculate the reliability score of each of the metrics. Consequently, Ochiai had a reliability score of 66,67 %, Barinel and Jaccard 61,90 %, Tarantula 57,14 % and lastly, D* and Op2 52,38 %. 14,29 % is the difference between the best and the worst reliability score value.

	Barinel	**D***	**Jaccard**	**Ochiai**	**Op2**	**Tarantula**
Total patches found	13	11	13	14	11	12
Reliability score (%)	61,90	52,38	61,90	66,67	52,38	57,14

Table 5.3: Reliability summary.

5.3 Suspiciousness correctness

If we now turn to the suspiciousness correctness aspect, the metric that performed best was Barinel, with 1007,41 points in total. This result was followed by: Jaccard (1000,35), i.e., only 7,06 points less then Barinel; Ochiai and Op2 728,52 and 731,14 points, respectively; Tarantula - 485,68 and D* 45,17 points. From the data presented in Table 5.4, it can be seen that in 92 out of 126, i.e., in 73 % of cases, a suspiciousness correctness score was successfully assigned. This fact indicates that in 92 cases, a buggy statement was identified by a metric and therefore included in the list of suspicious code (suspicious_math-*numberoftest*.json). If we take a further look at this aspect, then we realize that Ochiai was the metric that identified the buggy statement in 16 out of 21 cases, i.e, in 76 % of cases and thus performed the best. Nonetheless, the remaining metrics (Barinel, D*, Jaccard, Op2 and Tarantula) included the buggy statement in their suspiciousness lists in 15 out of 21 cases (71 %). In this context the difference between the two results is only of 5 percentage points.

Bug Id	Barinel	D*	Jaccard	Ochiai	Op2	Tarantula
Math 2	3,37	1,48	3,37	2,97	3,48	2,97
Math 5	0,00	1,58	0,00	0,00	0,00	0,00
Math 7	0,00	1,38	1,84	1,84	1,84	1,84
Math 8	32,33	2,57	32,33	24,25	24,25	24,25
Math 20	0,00	0,00	0,00	0,00	0,00	0,00
Math 32	394,00	0,00	394,00	394,00	394,00	394,00
Math 39	8,57	2,88	8,57	8,57	8,57	8,57
Math 40	80,00	1,58	80,00	160,00	160,00	2,39
Math 44	0,00	0,00	0,00	0,00	0,00	0,00
Math 64	8,90	0,00	0,00	1,69	0,00	0,00
Math 70	0,00	1,74	0,00	0,00	0,00	0,00
Math 71	1,01	1,36	1,01	1,30	1,84	3,31
Math 73	0,00	0,00	0,00	0,00	0,00	0,00
Math 74	2,24	1,79	2,24	2,24	2,24	2,24
Math 78	2,65	1,54	2,65	2,65	2,65	2,65
Math 80	31,71	6,51	31,71	31,71	31,71	5,04
Math 81	382,00	5,14	382,00	47,75	63,67	1,53
Math 82	5,17	2,26	5,17	5,17	5,17	5,17
Math 84	13,63	8,57	13,63	13,63	13,63	13,63
Math 85	28,33	2,45	28,33	28,33	1,89	1,89
Math 95	13,50	2,33	13,50	2,43	16,20	16,20
Suspiciousness correctness (total)	**1007,41**	**45,17**	**1000,35**	**728,52**	**731,14**	**485,68**

Table 5.4: Suspiciousness correctness values of the chosen fault localization metrics.

Chapter 6

Discussion and conclusions

As stated in the introductory chapter of the thesis, debugging code is a resource and time-consuming activity, and the ongoing research in the APR field aims to address this problem. In this study we tried to elucidate the role of suspiciousness metrics for fault localization, as the fault localization step is a critical part of the APR workflow of APR tools such as ASTOR. Furthermore, we studied the impact of distinct metrics for the computation of suspiciousness by conducting an experiment. We aimed to measure the impact of distinct metrics for fault localization on the performance of ASTOR by exploring the execution time of ASTOR, the reliability of the chosen metrics, and the notion of suspiciousness correctness. Our experiment showed that the execution time of ASTOR until the first plausible patch was found, could variate based on which metric was used for the fault localization step. The most significant difference between ASTOR executions regards the tests performed on the bug Math 82, where the difference between the fastest and slowest execution was of 553,23 s (the slowest execution was 571,31 s, i.e., +3060 % to the fastest execution which was 18,08 s). The experiment showed also that the mean execution time for the cases when a plausible patch was found could differ from metric to metric. In this context, Barinel resulted as the metric for which the executions of ASTOR were fastest (mean value). Based on the results, we could establish that the mean reliability score for the chosen metrics was 58,73 %, and the best score of 66,67 % of reliability was obtained by Ochiai. As it comes to the aspect of suspiciousness correctness, the best score was obtained by Barinel (1007,41 points). The three aforementioned aspects are summarized by the graphs in Figure 6.1. As the metrics that had the fastest mean execution time had also a relatively high suspiciousness correctness score, there exist a possibility of correlation between them. This conclusion seems to resonate with the finindgs of Liu et al. that inaccurate fault localization has a real impact on the performance of an automatic repair attempt. We believe that the experimental results brought valuable insights regarding the influence of fault localization metrics on the effectiveness of ASTOR.

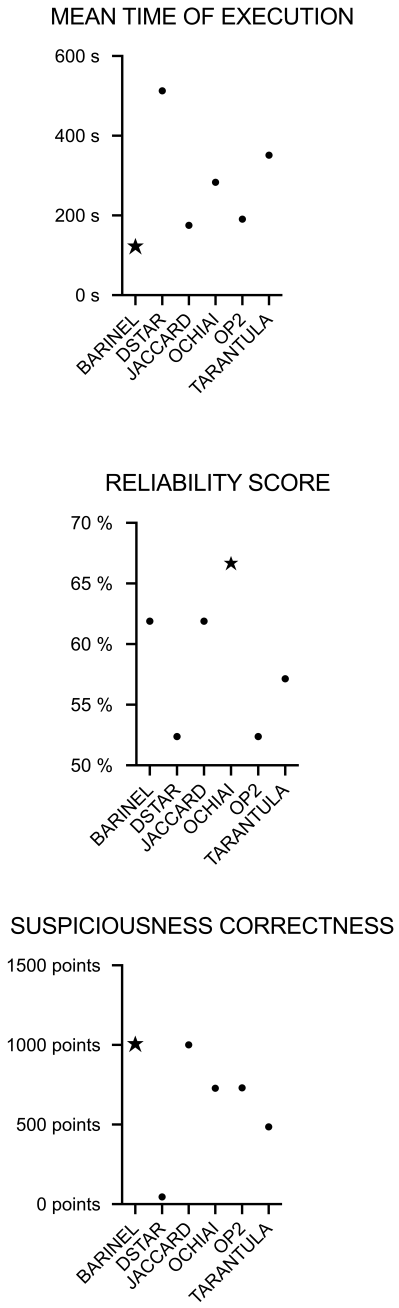

Figure 6.1: Summary of the collected results. The metric performing best is marked with a star symbol. The presented data relates to the aspects discussed in Section 4.6.

6.1 Validity and reliability

The generalisability of our findings may be limited to the scope of the APR tool and the conducted experiment. To enhance the study's validity and reliability, we are transparent with how the research was conducted. The code for the chosen fault localization metrics, the shell scripts executed to run the tests and the obtained results are publicly available at a GitHub repository (https://github.com/mafaust/aprthesis). To further address the question of the measurements' reliability, during the experiment 60 executions of ASTOR, i.e., 10 per metric, for the Defects4J bug Math 95 were analyzed and are available as a supplement of the thesis. An important remark has to be made in regards to the calculation of suspiciousness correctness (Section 4.6), i.e., a concept and a formula created for the purpose of this study. The values calculated with a non-standardized evaluation metric such as this one have to be interpreted with caution, as their validity outside the scope of the study can be limited.

6.2 Limitations

Our study was limited to one selected APR tool written in Java and for Java code exclusively, six selected metrics and a selection of bugs from the Defects4J dataset. The criteria used for the aforementioned selections are documented in the introduction of the thesis and in the experimental framework and methodology chapter. It's relevant to note that ASTOR's time-out value (60 minutes) can also be seen as a limitation to the conducted experiment.

6.3 Ethical considerations

As the experiment focused on analyzing data obtained by executing ASTOR, no immediate ethical issues could be identified. Our experimentation didn't involve human subjects or sensitive data, nor did the used code and studied material (ASTOR source code, input, output and Defects4J code). Furthermore, during the investigation, no data of sensitive character was obtained.

6.4 Future research

In order to increase the generalisability of our findings much can be done in the future. On a general level, experimenting with other available APR tools would make it possible to verify our findings outside the scope of ASTOR. Furthermore, the use of inferential statistics on more substantial data could address also the issue of generalisability and could help establish if a correlation between performance and suspiciousness correctness exists. To increase the study's validity and reliability, it would be of great benefit to compare the data with external sources and other experiments conducted in a similar manner. Future research could benefit from conducting the experiment on a larger dataset, i.e., not only

on Defects4J bugs but on other available open-source and real-life projects. As a design modification it is worth considering to increase the time-out value in future studies, so that more time is given to find plausible patches. Last but not least, integrating the metrics with the existing ASTOR repository, so that they would be available via ASTOR extension points, would make the experimentation more straightforward.

Bibliography

[1] Abramson, D., Foster, I., Michalakes, J. and Sosic, R. [1995], Relative debugging and its application to the development of large numerical models, *in* 'Supercomputing'95: Proceedings of the 1995 ACM/IEEE Conference on Supercomputing', IEEE, pp. 51–51.

[2] Abreu, R., Zoeteweij, P., Golsteijn, R. and Van Gemund, A. J. [2009], 'A practical evaluation of spectrum-based fault localization', *Journal of Systems and Software* **82**(11), 1780–1792.

[3] Abreu, R., Zoeteweij, P. and Van Gemund, A. J. [2007], On the accuracy of spectrum-based fault localization, *in* 'Testing: Academic and industrial conference practice and research techniques-MUTATION (TAICPART-MUTATION 2007)', IEEE, pp. 89–98.

[4] Abreu, R., Zoeteweij, P. and Van Gemund, A. J. [2009], Spectrum-based multiple fault localization, *in* '2009 IEEE/ACM International Conference on Automated Software Engineering', IEEE, pp. 88–99.

[5] An, G., Yoon, J. and Yoo, S. [2021], Searching for multi-fault programs in defects4j, *in* U. O'Reilly and X. Devroey, eds, 'Search-Based Software Engineering - 13th International Symposium, SSBSE 2021, Bari, Italy, October 11-12, 2021, Proceedings', Vol. 12914 of *Lecture Notes in Computer Science*, Springer, pp. 153–158.
URL: *https://doi.org/10.1007/978-3-030-88106-1_11*

[6] Assi, R. A., Trad, C., Maalouf, M. and Masri, W. [2019], 'Coincidental correctness in the defects4j benchmark', *Softw. Test. Verification Reliab.* **29**(3).
URL: *https://doi.org/10.1002/stvr.1696*

[7] Britton, T., Jeng, L., Carver, G. and Cheak, P. [2013], 'Reversible debugging software "quantify the time and cost saved using reversible debuggers"'.

[8] Cellier, P., Ducassé, M., Ferré, S. and Ridoux, O. [2008], Formal concept analysis enhances fault localization in software, *in* R. Medina and S. A. Obiedkov, eds, 'Formal Concept Analysis, 6th International Conference,

ICFCA 2008, Montreal, Canada, February 25-28, 2008, Proceedings', Vol. 4933 of *Lecture Notes in Computer Science*, Springer, pp. 273–288.
URL: *https://doi.org/10.1007/978-3-540-78137-0_20*

[9] Cellier, P., Ducassé, M., Ferré, S. and Ridoux, O. [2011], Multiple fault localization with data mining, *in* 'Proceedings of the 23rd International Conference on Software Engineering & Knowledge Engineering (SEKE'2011), Eden Roc Renaissance, Miami Beach, USA, July 7-9, 2011', Knowledge Systems Institute Graduate School, pp. 238–243.

[10] Friedrich, G., Stumptner, M. and Wotawa, F. [1999], 'Model-based diagnosis of hardware designs', *Artif. Intell.* **111**(1-2), 3–39.
URL: *https://doi.org/10.1016/S0004-3702(99)00034-X*

[11] Gay, G. and Just, R. [2020], Defects4j as a challenge case for the search-based software engineering community, *in* A. Aleti and A. Panichella, eds, 'Search-Based Software Engineering - 12th International Symposium, SSBSE 2020, Bari, Italy, October 7-8, 2020, Proceedings', Vol. 12420 of *Lecture Notes in Computer Science*, Springer, pp. 255–261.
URL: *https://doi.org/10.1007/978-3-030-59762-7_19*

[12] Gazzola, L., Micucci, D. and Mariani, L. [2019], 'Automatic software repair: A survey', *IEEE Trans. Software Eng.* **45**(1), 34–67.
URL: *https://doi.org/10.1109/TSE.2017.2755013*

[13] Goues, C. L., Pradel, M. and Roychoudhury, A. [2019], 'Automated program repair', *Commun. ACM* **62**(12), 56–65.
URL: *https://doi.org/10.1145/3318162*

[14] Goues, C. L., Pradel, M., Roychoudhury, A. and Chandra, S. [2021], 'Automatic program repair', *IEEE Softw.* **38**(4), 22–27.
URL: *https://doi.org/10.1109/MS.2021.3072577*

[15] Jones, J. A. and Harrold, M. J. [2005], Empirical evaluation of the tarantula automatic fault-localization technique, *in* D. F. Redmiles, T. Ellman and A. Zisman, eds, '20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005), November 7-11, 2005, Long Beach, CA, USA', ACM, pp. 273–282.
URL: *https://doi.org/10.1145/1101908.1101949*

[16] Just, R., Jalali, D. and Ernst, M. D. [2014], Defects4j: a database of existing faults to enable controlled testing studies for java programs, *in* C. S. Pasareanu and D. Marinov, eds, 'International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014', ACM, pp. 437–440.
URL: *https://doi.org/10.1145/2610384.2628055*

[17] Liu, K., Koyuncu, A., Bissyandé, T. F., Kim, D., Klein, J. and Traon, Y. L. [2019], You cannot fix what you cannot find! an investigation of fault

localization bias in benchmarking automated program repair systems, *in* '12th IEEE Conference on Software Testing, Validation and Verification, ICST 2019, Xi'an, China, April 22-27, 2019', IEEE, pp. 102–113.
URL: *https://doi.org/10.1109/ICST.2019.00020*

[18] Martinez, M., Durieux, T., Sommerard, R., Xuan, J. and Monperrus, M. [2018], 'Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset', *CoRR* **abs/1811.02429**.
URL: *http://arxiv.org/abs/1811.02429*

[19] Martinez, M. and Monperrus, M. [2019], 'Astor: Exploring the design space of generate-and-validate program repair beyond genprog', *J. Syst. Softw.* **151**, 65–80.
URL: *https://doi.org/10.1016/j.jss.2019.01.069*

[20] Mateis, C., Stumptner, M. and Wotawa, F. [2000], Modeling java programs for diagnosis, *in* W. Horn, ed., 'ECAI 2000, Proceedings of the 14th European Conference on Artificial Intelligence, Berlin, Germany, August 20-25, 2000', IOS Press, pp. 171–175.

[21] Monperrus, M. [2018*a*], 'Automatic software repair: A bibliography', *ACM Comput. Surv.* **51**(1), 17:1–17:24.
URL: *https://doi.org/10.1145/3105906*

[22] Monperrus, M. [2018*b*], The living review on automated program repair, Technical Report hal-01956501, HAL/archives-ouvertes.fr.

[23] Naish, L., Lee, H. J. and Ramamohanarao, K. [2011], 'A model for spectra-based software diagnosis', *ACM Transactions on software engineering and methodology (TOSEM)* **20**(3), 1–32.

[24] Pearson, S., Campos, J., Just, R., Fraser, G., Abreu, R., Ernst, M. D., Pang, D. and Keller, B. [2017], Evaluating and improving fault localization, *in* S. Uchitel, A. Orso and M. P. Robillard, eds, 'Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017', IEEE / ACM, pp. 609–620.
URL: *https://doi.org/10.1109/ICSE.2017.62*

[25] Rothenberg, B. and Grumberg, O. [2020], Must fault localization for program repair, *in* S. K. Lahiri and C. Wang, eds, 'Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II', Vol. 12225 of *Lecture Notes in Computer Science*, Springer, pp. 658–680.
URL: *https://doi.org/10.1007/978-3-030-53291-8_33*

[26] Silva, A., Martinez, M., Danglot, B., Ginelli, D. and Monperrus, M. [2021], 'FLACOCO: fault localization for java based on industry-grade coverage', *CoRR* **abs/2111.12513**.
URL: *https://arxiv.org/abs/2111.12513*

[27] Sobreira, V., Durieux, T., Madeiral, F., Monperrus, M. and de Almeida Maia, M. [2018], Dissection of a bug dataset: Anatomy of 395 patches from defects4j, *in* R. Oliveto, M. D. Penta and D. C. Shepherd, eds, '25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018', IEEE Computer Society, pp. 130–140.
URL: *https://doi.org/10.1109/SANER.2018.8330203*

[28] Tufano, M., Watson, C., Bavota, G., Penta, M. D., White, M. and Poshyvanyk, D. [2018], An empirical investigation into learning bug-fixing patches in the wild via neural machine translation, *in* M. Huchard, C. Kästner and G. Fraser, eds, 'Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018', ACM, pp. 832–837.
URL: *https://doi.org/10.1145/3238147.3240732*

[29] Urli, S., Yu, Z., Seinturier, L. and Monperrus, M. [2018], How to design a program repair bot?: insights from the repairnator project, *in* F. Paulisch and J. Bosch, eds, 'Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2018, Gothenburg, Sweden, May 27 - June 03, 2018', ACM, pp. 95–104.
URL: *https://doi.org/10.1145/3183519.3183540*

[30] Weiser, M. D. [1979], *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method*, University of Michigan.

[31] Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C. and Regnell, B. [2012], *Experimentation in Software Engineering*, Springer.
URL: *https://doi.org/10.1007/978-3-642-29044-2*

[32] Wong, W. E., Debroy, V. and Choi, B. [2010], 'A family of code coverage-based heuristics for effective fault localization', *J. Syst. Softw.* **83**(2), 188–208.
URL: *https://doi.org/10.1016/j.jss.2009.09.037*

[33] Wong, W. E., Debroy, V., Gao, R. and Li, Y. [2013], 'The dstar method for effective software fault localization', *IEEE Transactions on Reliability* **63**(1), 290–308.

[34] Wong, W. E., Debroy, V., Golden, R., Xu, X. and Thuraisingham, B. [2012], 'Effective software fault localization using an RBF neural network', *IEEE Trans. Reliab.* **61**(1), 149–169.
URL: *https://doi.org/10.1109/TR.2011.2172031*

[35] Wong, W. E., Gao, R., Li, Y., Abreu, R. and Wotawa, F. [2016], 'A survey on software fault localization', *IEEE Trans. Software Eng.* **42**(8), 707–740.
URL: *https://doi.org/10.1109/TSE.2016.2521368*

[36] Wong, W. E. and Qi, Y. [2009], 'Bp neural network-based effective fault localization', *Int. J. Softw. Eng. Knowl. Eng.* **19**(4), 573–597.
URL: *https://doi.org/10.1142/S0218194009004026X*

[37] Wotawa, F. [2002], 'On the relationship between model-based debugging and program slicing', *Artif. Intell.* **135**(1-2), 125–143.
URL: *https://doi.org/10.1016/S0004-3702(01)00161-8*

[38] Xuan, J., Martinez, M., Demarco, F., Clement, M., Marcote, S. R. L., Durieux, T., Berre, D. L. and Monperrus, M. [2017], 'Nopol: Automatic repair of conditional statement bugs in java programs', *IEEE Trans. Software Eng.* **43**(1), 34–55.
URL: *https://doi.org/10.1109/TSE.2016.2560811*

[39] Yang, D., Liu, K., Kim, D., Koyuncu, A., Kim, K., Tian, H., Lei, Y., Mao, X., Klein, J. and Bissyandé, T. F. [2021], 'Where were the repair ingredients for defects4j bugs?', *Empir. Softw. Eng.* **26**(6), 122.
URL: *https://doi.org/10.1007/s10664-021-10003-7*

[40] Yu, Z., Hu, H., Bai, C., Cai, K. and Wong, W. E. [2011], GUI software fault localization using n-gram analysis, *in* T. M. Khoshgoftaar, ed., '13th IEEE International Symposium on High-Assurance Systems Engineering, HASE 2011, Boca Raton, FL, USA, November 10-12, 2011', IEEE Computer Society, pp. 325–332.
URL: *https://doi.org/10.1109/HASE.2011.29*

[41] Zhang, X., Gupta, N. and Gupta, R. [2006], Locating faults through automated predicate switching, *in* L. J. Osterweil, H. D. Rombach and M. L. Soffa, eds, '28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006', ACM, pp. 272–281.
URL: *https://doi.org/10.1145/1134285.1134324*

[42] Zimmermann, T. and Zeller, A. [2001], Visualizing memory graphs, *in* S. Diehl, ed., 'Software Visualization, International Seminar Dagstuhl Castle, Germany, May 20-25, 2001, Revised Lectures', Vol. 2269 of *Lecture Notes in Computer Science*, Springer, pp. 191–204.
URL: *https://doi.org/10.1007/3-540-45875-1_15*

Appendix A

Java code of the chosen FL metrics

The following Java classes are the implementation of the six SBFL metrics chosen for the experimentation phase of the project. The metrics, i.e., Barinel, D*, Jaccard, Ochiai, Op2 and Tarantula implement the FLACOCO *Formula* interface. In five formulas an if-statement is added to prevent division by zero.

A.1 Barinel

```
1  package fr.spoonlabs.flacoco.localization.spectrum.formulas;
2
3  public class BarinelFormula implements Formula {
4
5      public BarinelFormula() {
6      }
7
8      @Override
9      public double compute(
10     int nPassingNotExecuting, int nFailingNotExecuting,
11     int nPassingExecuting, int nFailingExecuting) {
12         if(nPassingExecuting + nFailingExecuting == 0) {
13             return 0;
14         }
15
16         return 1.0 - ((double)nPassingExecuting /
17         (double)(nPassingExecuting + nFailingExecuting));
18     }
19 }
```

A.2 D*

```
package fr.spoonlabs.flacoco.localization.spectrum.formulas;

public class DStarFormula implements Formula {

    public DStarFormula() {
    }

    @Override
    public double compute(
    int nPassingNotExecuting, int nFailingNotExecuting,
    int nPassingExecuting, int nFailingExecuting) {

        if((nFailingNotExecuting + nPassingExecuting)== 0 ||
        (nFailingExecuting == 0)) {
            return 0;
        }

        return Math.pow(nFailingExecuting, 2) /
        (double)(nFailingNotExecuting + nPassingExecuting);
    }
}
```

A.3 Jaccard

```
package fr.spoonlabs.flacoco.localization.spectrum.formulas;

public class JaccardFormula implements Formula {

    public JaccardFormula() {
    }

    @Override
    public double compute(
    int nPassingNotExecuting, int nFailingNotExecuting,
    int nPassingExecuting, int nFailingExecuting) {

        if ((nFailingExecuting + nFailingNotExecuting +
        nPassingExecuting) == 0) {
            return 0;
        }

        return (double)nFailingExecuting /
        (double)(nFailingExecuting + nFailingNotExecuting +
        nPassingExecuting);
    }
}
```

A.4 Ochiai

```java
package fr.spoonlabs.flacoco.localization.spectrum.formulas;

public class OchiaiFormula implements Formula {

    public OchiaiFormula() {
    }

    @Override
    public double compute(
    int nPassingNotExecuting, int nFailingNotExecuting,
    int nPassingExecuting, int nFailingExecuting) {

        if ((nFailingExecuting + nPassingExecuting == 0) ||
        (nFailingExecuting + nFailingNotExecuting == 0)) {
            return 0;
        }

        return nFailingExecuting /
        (Math.sqrt((nFailingExecuting + nFailingNotExecuting) *
        (nFailingExecuting + nPassingExecuting)));
    }
}
```

A.5 Op2

```java
package fr.spoonlabs.flacoco.localization.spectrum.formulas;

public class Op2Formula implements Formula{

    public Op2Formula() {
    }

    @Override
    public double compute(
    int nPassingNotExecuting, int nFailingNotExecuting,
    int nPassingExecuting, int nFailingExecuting) {
        return (double)nFailingExecuting -
        ((double)nPassingExecuting /
        ((nPassingExecuting + nPassingNotExecuting) + 1.0));
    }
}
```

A.6 Tarantula

```
package fr.spoonlabs.flacoco.localization.spectrum.formulas;

public class TarantulaFormula implements Formula{

    public TarantulaFormula(){
    }

    @Override
    public double compute(
    int nPassingNotExecuting, int nFailingNotExecuting,
    int nPassingExecuting, int nFailingExecuting) {
        final int nTotalFailing =
        nFailingExecuting + nFailingNotExecuting;
        final int nTotalPassing =
        nPassingExecuting + nPassingNotExecuting;
        if ((nTotalFailing == 0) || (nTotalPassing == 0) ||
        ((((double)nFailingExecuting / (double)nTotalFailing) +
        ((double)nPassingExecuting / (double)nTotalPassing)) ==
        0.0)){
            return 0;
        }

        return ((double)nFailingExecuting /
        (double)(nFailingExecuting + nFailingNotExecuting)) /
        (((double)nFailingExecuting / (double)(nFailingExecuting +
        nFailingNotExecuting)) + ((double)nPassingExecuting /
        (double)(nPassingExecuting + nPassingNotExecuting)));
    }
}
```

Appendix B

Measurements' reliability check (Math 95)

The following plots illustrate the reliability check performed on the Defects4J bug Math 95. The execution time of ASTOR with a given metric is represented as a dot. Furthermore, the mean execution time and the standard deviation are included in the figures.

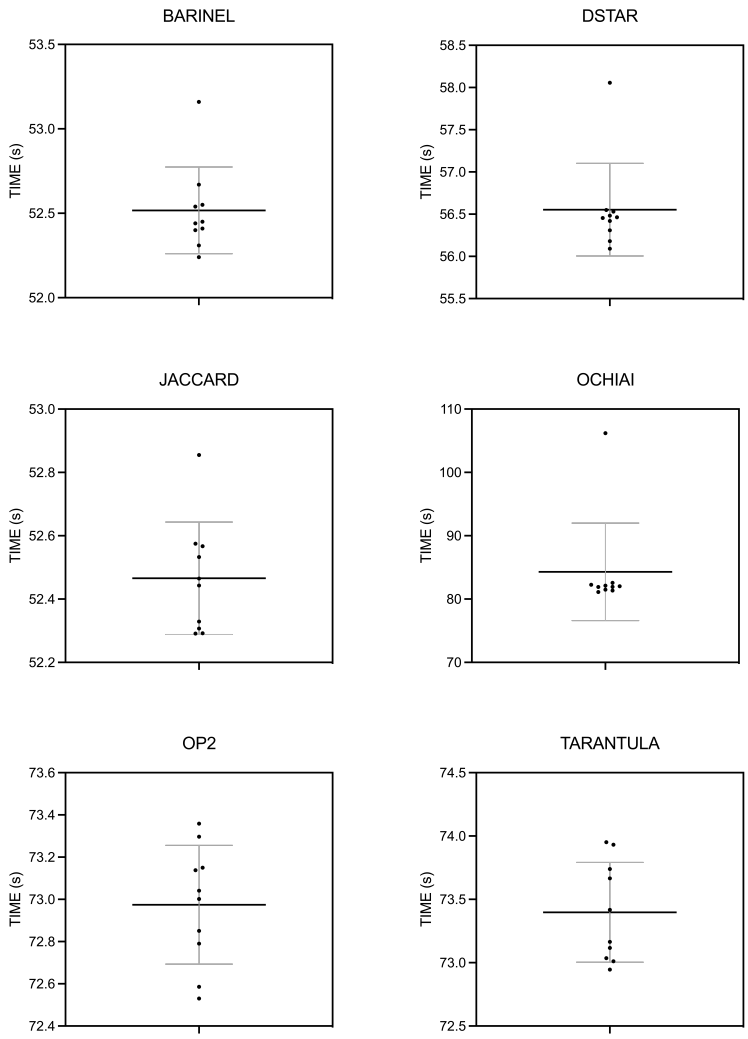

Figure B.1: Measurements' reliability check - Math 95.