D 2018

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

SPECTRUM-BASED DIAGNOSIS: MEASUREMENTS, IMPROVEMENTS AND APPLICATIONS

ALEXANDRE CAMPOS PEREZ
TESE DE DOUTORAMENTO APRESENTADA
À FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO EM
ENGENHARIA INFORMÁTICA

U.PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Spectrum-based Diagnosis: Measurements, Improvements and Applications

Alexandre Perez

Supervisor: Rui Maranhão Abreu

Doctoral Program in Informatics Engineering

July, 2018

© Alexandre Perez: July, 2018

Spectrum-based Diagnosis: Measurements, Improvements and Applications

Alexandre Perez

Thesis submitted to the Faculty of Engineering of University of Porto to obtain the degree of

Doctor of Philosophy in Informatics Engineering

President: Professor Carlos Soares
University of Porto, Portugal

Referee: Professor Franz Wotava
Graz University of Technology, Austria

Referee: Professor João Saraiva
University of Minho, Portugal

Referee: Professor Nuno Nunes
IST, University of Lisbon, Portugal

Referee: Professor João Cardoso
University of Porto, Portugal

Referee: Professor João Pascoal Faria
University of Porto, Portugal

Supervisor: Professor Rui Maranhão Abreu
IST, University of Lisbon, Portugal

July, 2018

The work developed in this thesis was supported by the scholarship number SFRH/BD/95339/2013 from Fundação para a Ciência e Tecnologia (FCT).

HASLab
INESCTEC

FCT
Fundação para a Ciência e a Tecnologia
MINISTÉRIO DA EDUCAÇÃO E CIÊNCIA

PD F
FCT PhD PROGRAMMES

POPH

QREN
QUADRO DE REFERÊNCIA ESTRATÉGICO NACIONAL

REPÚBLICA PORTUGUESA
CIÊNCIA, TECNOLOGIA E ENSINO SUPERIOR

UNIÃO EUROPEIA
Fundo Social Europeu

Abstract

Debugging—the process of locating and fixing abnormal behavior in software—is often cited as one of the most costly and unpredictable phases of developing software. It is therefore essential to minimize the impact of debugging in software development, while still ensuring the quality of software systems.

Previous research has proposed several automated fault localization techniques, that aid developers by pinpointing which software components are likely to be faulty. Among such techniques is **Spectrum-based Fault Localization (*SFL*)**. *SFL* is a runtime technique that collects the involvement of each software component in test cases—usually called *program spectrum*—, and reasons about their correlation to failing test outcomes. The assumption is that components frequently involved in failing tests are more likely to be faulty and, conversely, components more frequently involved in passing tests are less likely to be the root cause. The abstract nature of program spectra allows for a language-agnostic, lightweight analysis (compared to related approaches), with considerable accuracy. However, there are limited accounts of successful transitions of *SFL* into practice. In fact, studies show that developers quickly discard diagnostic reports after inspecting a small, limited number of fault candidates reported by the technique.

This thesis proposes several approaches aimed at enhancing the usefulness of *SFL* in practice. Namely, we introduce predictive measurements of diagnostic performance, improve fault comprehension and fault isolation, and apply *SFL* theory to the context of feature detection for program maintenance.

First, we propose a runtime measurement, named *DDU*, aimed at assessing the effectiveness of a test suite at diagnosing potential faults in the code. *DDU* measures three traits found in highly-diagnosable spectra, namely: moderate component involvement *density*, high *diversity* of test cases, and high component involvement *unambiguity*. Through these traits, *DDU* ensures that distinct combinations of components are exercised in tandem to maximize the usefulness of *SFL* at pinpointing the cause of any error that may occur. The *DDU* diagnosability metric thereby serves as an indicator of the accuracy of *SFL*'s diagnostic reports for a given test suite—similarly to how adequacy measurements, such as branch coverage, act as indicators for fault detection—allowing users to measure, and also improve, the quality of their test suite.

Second, we conduct a large-scale evaluation assessing how faults are actually fixed in practice. This evaluation is motivated by the fact that similarity-based *SFL* techniques are most effective when only one fault is responsible for all test failures. Similarity-based techniques cannot handle multiple simultaneous faults with the same degree of accuracy, unlike the more computationally expensive reasoning-based *SFL* techniques. Our hypothesis is that, in practice, faults are mostly detected and fixed in isolation, thereby resulting in single-fault localization problems, which can diagnosed with lightweight similarity-based *SFL* variants. We propose a methodology for mining software repositories and classifying fixes according to the number of faults they address. Our evaluation found that 82% of all fixes were single-faulted, yielding high diagnostic accuracy when similarity-based variants were used.

Third, we propose an enhancement to *SFL* that leverages concepts from **Qualitative Reasoning (*QR*)**. *QR* is an area of research within Artificial Intelligence that studies ways to abstract complexity by partitioning continuous-valued variables into discrete qualitative states, which are subsequently used to model systems in a more lightweight, tractable manner. Similarly, our approach—named *Q-SFL*—partitions the runtime value of spectrum components into sets of qualitative states. These states are then considered as contextual *SFL* components—their involvement in tests are thus recorded in the spectrum. The main advantages of such augmentation are increased fault isolation and improved fault comprehension. Our evaluation shows that augmenting *SFL* through qualitative partitioning can improve diagnostic accuracy, but further work is needed to develop effective automated partitioning strategies.

Lastly, as a way to expand the applicability of *SFL*, we propose **Spectrum-based Feature Comprehension (*SFC*)**. *SFC* provides a mapping of *SFL* concepts to the task of feature detection, which typically is the most time consuming step during software maintenance scenarios. *SFC* shares many similarities with *SFL*, but instead of correlating component coverage with failing tests, components are correlated with feature involvement. Our user study shows that users were able to more accurately pinpoint features with the aid of *SFC*, compared to using a test coverage tool. Furthermore, in cases where a mapping between tests and the features they exercise is not readily available (or if there are no tests at all), we propose **Participatory Feature Detection (*PFD*)**. *PFD* allows users to manually label their interactions with a system as associated or dissociated with the feature they want to locate. Our evaluation of *PFD* shows that the technique is able to achieve considerable accuracy at detecting features, even when users misclassify their recorded interactions.

Resumo

O *debugging*—processo de localizar e corrigir comportamento anormal em software—é frequentemente citado como uma das mais dispendiosas e imprevisíveis fases de desenvolvimento de software. É essencial, portanto, minimizar o impacto do *debugging*, embora garantindo ainda assim a qualidade dos sistemas de software.

A investigação anterior propôs várias técnicas automatizadas de localização de falhas, com o intuito de ajudar os desenvolvedores a identificar componentes de software provavelmente defeituosos. Entre tais técnicas é de salientar o **Spectrum-based Fault Localization (*SFL*)**. O *SFL* é uma técnica *runtime* que recolhe o envolvimento de componentes de software em testes—normalmente chamado de *program spectrum*—, correlacionando-o com os resultados de testes faltosos. A suposição é de que componentes frequentemente envolvidos em testes faltosos são mais prováveis de conterem defeitos e, inversamente, componentes frequentemente envolvidos em testes que passam são menos prováveis. A natureza abstrata dos *program spectra* permite uma análise leve (em comparação com abordagens relacionadas) e agnóstica a linguagens, com considerável precisão. No entanto, existem escassos relatos de transições bem-sucedidas para a prática. Efetivamente, os estudos mostram que os desenvolvedores descartam rapidamente os diagnósticos reportados pelo *SFL* após inspecionar um número limitado de candidatos a falhas.

Esta tese propõe abordagens destinadas a melhorar a utilidade e eficácia do *SFL* na prática. Nomeadamente, introduzimos medidas preditivas de desempenho do diagnóstico, melhoramos a compreensão de falhas e o isolamento das mesmas, e aplicamos o *SFL* no contexto de deteção de *features* durante a fase de manutenção.

Primeiro, propomos uma métrica *runtime*, denominada de *DDU*, destinada a avaliar a eficácia de um conjunto de testes no diagnóstico de possíveis falhas no código. O *DDU* mede três caraterísticas encontradas em *spectra* altamente diagnosticáveis: a moderada *densidade* de envolvimento de componentes, a alta *diversidade* de casos de teste, e alta concentração de componentes *sem envolvimento ambíguo*. Através destas caraterísticas, o *DDU* garante que combinações distintas de componentes sejam exercitadas por forma a maximizar a utilidade do *SFL*. O *DDU* serve como um indicador da precisão dos relatórios do *SFL* para uma determinada conjunto de testes, permitindo medir e também melhorar a qualidade do mesmo.

Em segundo lugar, realizamos uma avaliação em larga escala sobre como as falhas são reparadas na prática. Esta avaliação é motivada pelo facto de que as técnicas *SFL* baseadas em similaridade são mais eficazes quando apenas um defeito é responsável por todas as falhas de teste. O *SFL* baseado em similaridade não localiza múltiplas falhas com o mesmo grau de precisão, ao contrário das técnicas *SFL* baseadas em *reasoning* (mais dispendiosas em termos computacionais). A nossa hipótese é que, na prática, a maioria dos defeitos são detetados e reparados isoladamente, resultando em problemas de localização de defeito único, adequados para variantes do *SFL* baseadas em similaridade. Propomos uma metodologia de *mining* de repositórios de software que identifica reparações de falhas e as classifica de acordo com o número de falhas que eliminam. A nossa avaliação constatou que 82% de todas as correções envolveram uma única falha, produzindo alta precisão de diagnóstico por parte de variantes baseadas em similaridade.

Em terceiro lugar, propomos uma melhoria para *SFL* que utiliza conceitos de **Qualitative Reasoning (*QR*)**. O *QR* é uma área de pesquisa da Inteligência Artificial que estuda formas de abstrair a complexidade de variáveis contínuas através do particionamento do seu valor em estados qualitativos discretos, posteriormente usados para modelação de sistemas. A nossa abordagem—denominada de *Q-SFL*—particiona o valor *runtime* de componentes do *spectrum* em conjuntos de estados qualitativos. Esses estados são então considerados como componentes *contextuais*, e o seu envolvimento nos testes é gravado no *spectrum*. As principais vantagens assentam na melhoria do isolamento de falhas e da sua compreensão. A nossa avaliação mostra que aumentar *spectra* através do particionamento qualitativo de componentes melhora a precisão do diagnóstico, mas é necessário mais trabalho para desenvolver estratégias de particionamento automatizadas.

Por fim, como forma de expandir a aplicabilidade das análises baseadas em *spectra*, propomos o **Spectrum-based Feature Comprehension (*SFC*)**. O *SFC* mapeia os conceitos do *SFL* para a tarefa de deteção de *features*, que normalmente é a etapa mais demorada durante a manutenção de software. O *SFC* partilha semelhanças com o *SFL*, mas em vez de correlacionar a cobertura de componentes com falhas de teste, a correlação é feita com o envolvimento de *features*. O nosso estudo mostra que os utilizadores são capazes de identificar *features* com mais precisão com o auxílio do *SFC*, em comparação com ferramentas de cobertura. Além disso, nos casos em que o mapeamento entre testes e *features* que eles exercitam não estão disponíveis, propomos o **Participatory Feature Detection (*PFD*)**. O *PFD* permite que os utilizadores classifiquem manualmente as suas interações com o sistema como associadas ou dissociadas à *feature* que querem localizar. A nossa avaliação mostra que o *PFD* é capaz de alcançar uma precisão considerável, mesmo quando os utilizadores classificam erroneamente as suas interações.

Acknowledgements

This thesis would certainly not have been possible without the help, support, and guidance of many people whom I would like to thank.

I'd like to start by thanking my advisor and friend Rui Maranhão Abreu. His advice and guidance were absolutely vital for the success of this work. I really admired this tenacity and optimism throughout this bumpy process. Thank you, Rui, for believing in me and in my work, even when I wouldn't!

I was fortunate enough to be able to collaborate with Arie van Deursen, from Delft University of Technology, and with Marcelo d'Amorim, from Federal University of Pernambuco. I want to thank them for taking time out of their busy schedules to kindly offer crucial and insightful feedback. I certainly hope to be ale to work with both of them again.

It is without a doubt that my tenure at Palo Alto Research Center has shaped me as a person and as a researcher. I'd like to express my utmost gratitude to Johan de Kleer for making it possible. While at PARC, I have met many wonderful people who went out of their way to help me in many different ways. Namely, I want to thank Danny Bobrow, Ion Matei, Joy Smith, Nora Boettcher, Jonathan Rubin, Alexander Feldman, Saigopal Nelaturi, Judy Farish, and Lara Crawford for their time, advice, and generosity. I am indebted to you all!

I want to thank everyone in the "Quasar" research group, particularly Luís Cruz, José Campos, and Nuno Cardoso, for all the interesting conversations (technical or otherwise), funny moments, and shared frustrations that inevitably happen to all PhD candidates.

To my friends and former classmates, João Santos, Luís Silva, Pedro Machado, Tiago Monteiro, and Francisco Silva, I thank their support and their levity. I am also thankful to my friends Fernando Sá, João Moura, and Eduardo Pais, who have, since our childhood, stood by my side through it all.

Last, but certainly not least, I would like to thank my parents, Jesus and Maria Filomena, for their tireless, unending support throughout my life. Would not have done it without you!

Contents

List of Figures

List of Tables

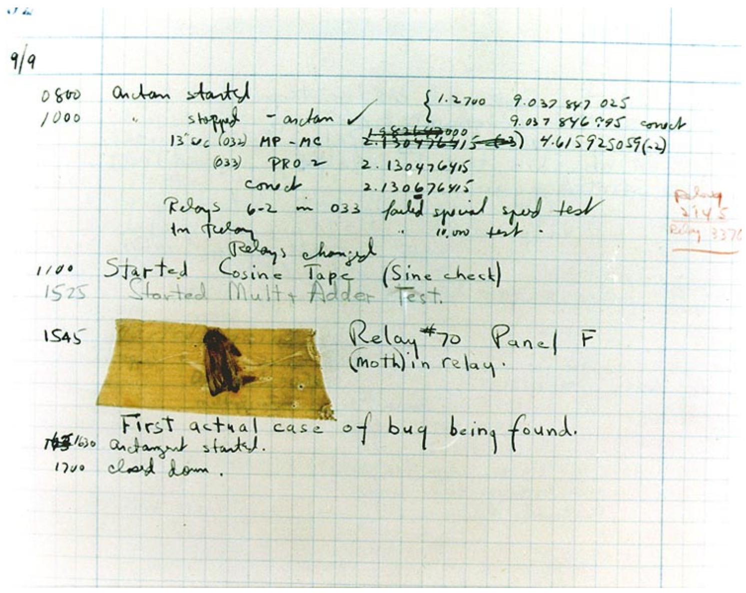
List of Acronyms

DDU	Density-Diversity-Uniqueness
GA	Genetic Algorithm
GUI	Graphical User Interface
MHS	Minimal Hitting Set
MLE	Maximum Likelihood Estimation
MSD	Model-based Software Debugging
PFD	Participatory Feature Detection
Q-SFL	Qualitative Spectrum-based Fault Localization
QR	Qualitative Reasoning
SFC	Spectrum-based Feature Comprehension
SFL	Spectrum-based Fault Localization
SPL	Software Product Lines

Introduction | 1

> *There are two ways to write error-free programs; only the third works.*
>
> — **Alan J. Perlis**

In 1947, the Harvard Mark II was being tested by Grace Murray Hopper and her associates when the machine suddenly stopped. Upon inspection, the error was traced to a dead moth that was trapped in a relay and had shorted out some of the circuits. The insect was removed and taped to the machine's logbook (see Figure 1.1) [Kidwell, 1998]. This incident is believed to have coined the use of the terms "bug", "debug" and "debugging" in the field of computer science. Since then, the term debugging is associated to the process of detecting, locating and fixing faulty behavior in computer programs.



9/9

0800 arctan started
1000 " stopped - arctan ✓ {1.2700 9.037 847 025
9.037 846 995 const
13" uc (032) MP - MC ~~1.982147000~~ ~~2.130476415~~ (-3)) 4.615925059(-2)
(033) PRO 2 2.130476415
const 2.130676415
Relays 6-2 in 033 failed special speed test
in relay " 10,000 test.
Relays changed
1100 Started Cosine Tape (Sine check)
1525 Started Mult + Adder Test.
1545 Relay #70 Panel F
(moth) in relay.
First actual case of bug being found.
1630 arctangent started.
1700 closed down.

Relay 2145
Relay 3370

Figure 1.1: First actual case of bug being found[1].

Software engineers have always faced the challenge of pinpointing and eliminating bugs in software—which frequently incurs in significant added costs during development [Tassey, 2002]. As readers may be aware, the typical flow of software development is generally comprised of four phases: (1) a requirements and design phase, (2) an implementation phase, (3) a testing and debugging phase, and finally

[1]U.S. Naval Historical Center Online Library Photograph NH 96566-KN.

(4) the release. While most of the aforementioned steps can be planned, estimated and executed with a considerably high degree of certainty, the same cannot be said for the testing and debugging phase. Besides the actual effort to locate the root cause of unintended behavior, certain defects require changing the application's implementation or, more intrusively, its design (therefore introducing loops in the development process). Studies have found debugging tasks to range from **50%** to **75%** of the total development cost [Hailpern and Santhanam, 2002].

One cannot estimate with a high degree of certainty the cost of debugging tasks (both in terms of time and money), as the effort required to find faults fluctuates wildly. This is due to the fact that the process of detecting, locating and fixing faults in the source code is not trivial and is error-prone. Even experienced developers are wrong almost 90% of the time in their initial guess while trying to identify the cause of a behavior that deviates from the intended one [Ko and Myers, 2008]. For this reason, it is important to minimize the impact that debugging has in the development process.

If program verification and debugging tasks are not thoroughly conducted, even bigger costs may arise. For instance, a landmark study performed in 2002 indicated that software defects constitute an annual $60 billion cost to the US economy alone [Tassey, 2002]. More recently, a 2013 study from Cambridge University estimates that software bugs cost the global software industry $316 billion per year[2]. It is essential to find ways to minimize the software debugging impact on a project's resources. However, it is imperative that the software quality (*i.e.*, its correctness) is not compromised. While some defects can be tolerated by users (or even not perceived at all), others may cause severe financial or life-threatening consequences. Infamous examples of drastic consequences caused by software defects are:

- The software malfunction of the rocket Ariane 5, which caused it to disintegrate 37 seconds after its launch [Lions, 1996; Dowson, 1997];
- The crash of a British Royal Air Force Chinook helicopter due to a software defect in the engine control computer, killing 29 people [Rogerson, 2002];
- The orbital insertion of the Mars Climate Orbiter probe due to a malfunction of its flight software [Stephenson et al., 1999];
- Apple's "goto fail" bug in the implementation of the SSL/TLS stack in iOS 6 and OSX 10.9, which caused the Server Key Exchange message not to be checked, potentially allowing man-in-the-middle attackers to spoof SSL servers [Bland, 2014];

[2]Research by Cambridge MBAs for tech firm Undo finds software bugs cost the industry $316 billion a year: `https://goo.gl/mikn7P` (accessed May 2018).

- OpenSSL's "heartbleed" bug which omitted a bounds check in the payload of `HeartbeatRequest` messages, allowing attackers to read arbitrary memory from peers [Durumeric et al., 2014];
- The Parity Library's Ethereum *smart contract* did not properly initialize the contract's wallet. A subsequent (possibly malicious) transaction initialized the wallet, and rendered approximately $280M worth of Ethereum cryptocurrency tokens unusable [Nikolic et al., 2018].

Debugging is then an important step that should not be disregarded when developing software. However, this activity consumes large amounts of resources. Therefore, ways to help developers in these tasks are continuously being researched. Nowadays, automatic fault localization techniques can aid developers/testers in pinpointing the root cause of software failures, and thereby reducing the debugging effort. For instance, coverage-based *runtime* analyses to automate the debugging process (referred to as **Spectrum-based Fault Localization (*SFL*)** techniques) have been proposed [Abreu et al., 2009d]. *SFL* techniques provide to developers a sorted list of likely faulty source-code locations, helping them prioritize their inspection of the program. By making use of program traces, these tools can abstract the complexity of the program, rendering them able to perform very lightweight (and even language-agnostic) analyses. Other, more intricate approaches have also been proposed, such as **Model-based Software Debugging (*MSD*)** [Mayer and Stumptner, 2003]. These techniques achieve higher diagnostic accuracy by inferring a model through the application of static and dynamic analyses of the program, and the use of logic reasoning to devise an explanation for the faulty behavior. These techniques, however, do not scale and their use in large, real-world projects is impractical [Mayer and Stumptner, 2008].

It is therefore our objective in this thesis to not only improve the accuracy of diagnostic techniques that are able to scale to real-world applications (*i.e*, *SFL*), but also guarantee their effectiveness and usefulness through predictive measurements and expanded scope of applications.

1.1 Concepts and Definitions

Throughout this document, we use the following definitions:

Definition 1 (Component) *A component is a software artifact that symbolizes a unit of computation and therefore is considered to be atomic*[3].

Definition 2 (Program/System) *A software program* Π *is formed by a sequence of one or more components.*

[3]Components can be considered at arbitrary granularity such as a class, a method, a statement, or a branch [Harrold et al., 1998].

Definition 3 (Test case) *A test case t is a (i, o) tuple, where i is a collection of input settings or variables for determining whether a software system works as expected or not, and o is the expected output. If $\Pi(i) = o$ the test case passes, otherwise fails. We also refer to test cases as system transactions.*

Definition 4 (Test suite) *A test suite $T = \{t_1, \cdots, t_N\}$ is a collection of test cases that are intended to test whether the program follows the specified set of requirements. The cardinality of T is the number of test cases in the set $|T| = N$.*

We also use the terminology adopted by Avižienis et al. [2004] to describe failures, errors and faults:

- A *failure* is an event that occurs when delivered service deviates from correct service.
- An *error* is a system state that may cause a failure.
- A *fault* (defect/bug) is the cause of an error in the system. **Faults**, **defects** and **bugs** are used interchangeably in this document.

In this thesis, we apply this terminology to software programs, where faults are bugs in the program code. Failures and errors are symptoms caused by faults in the program. The purpose of fault localization is to pinpoint the root cause of observed symptoms. Once a fault is identified, the following step is to *fix* it:

Definition 5 (Fix) *The set of source code modifications that, when applied, eliminate faults from the system.*

1.2 Spectrum-based Fault Localization (*SFL*)

SFL is a dynamic software diagnosis technique that exploits coverage information from both nominal and failing system executions to pinpoint the root cause of failures [Abreu et al., 2009d]. *SFL*'s main advantage is the fact that it reasons about the involvement of components in tests in an abstract way, allowing for a lightweight, flexible, and language-agnostic diagnostic analysis. In *SFL*, the following is given:

- A finite set $\mathcal{C} = \{c_1, c_2, \cdots, c_M\}$ of M system components;
- A finite set $\mathcal{T} = \{t_1, t_2, \cdots, t_N\}$ of N system transactions, which can be seen as records of a system execution, such as, *e.g.*, test cases;
- The outcome of system transactions is encoded in the error vector $e = \{e_1, e_2, \cdots, e_N\}$, where $e_j = 1$ if transaction t_j has failed and $e_j = 0$ otherwise;
- A $M \times N$ activity matrix $\mathcal{A}$, where $\mathcal{A}_{ij} = 1$ encodes the involvement of component c_i in transaction t_j, and $\mathcal{A}_{ij} = 0$ otherwise.

	t_1	t_2	t_3	t_4
c_1	●		●	●
c_2	●	●		
c_3		●		●
e	✗	✗	✗	✓

(a) Component coverage representation.

	t_1	t_2	t_3	t_4
c_1	1	0	1	1
c_2	1	1	0	0
c_3	0	1	0	1
e	1	1	1	0

(b) Matrix representation.

Figure 1.2: Spectrum of a system with 3 components and 4 transactions.

The pair $(\mathcal{A}, e)$ is commonly referred to as *spectrum* [Harrold et al., 1998]. Component involvement is typically gathered by injecting the program with probes that register the execution of each component at *runtime*, similar to the instrumentation one would find in coverage-gathering tools. An example spectrum is shown in Figure 1.2. This spectrum depicts four transactions (*i.e.*, test executions) of a system composed of three components. Transactions t_1, t_2 and t_3 fail, whereas in t_4 no error was observed. Note that the two subfigures—Figures 1.2a and 1.2b—depict the same spectrum, via two representations which are used throughout the paper: while the former emphasizes component involvement (the symbol ● is used when a component was active in a given transaction) and test outcomes (symbols ✗ and ✓ represent failing and passing tests, respectively), and is typically shown alongside code snippets; the latter, a more matrix-like representation, is used to depict the mathematical properties of the activity matrix and the error vector.

1.2.1 Similarity-based *SFL*

We now describe the classical, similarity-based approach to *SFL*. This approach assumes that the more a component's involvement resembles failing test outcomes, the more likely that component is the faulty one. Therefore, with the spectrum as input, the approach measures the resemblance of each row of the activity matrix $\mathcal{A}$ with the error vector e. For that, an intermediate component frequency aggregator is computed:

$$n_{pq}(i) = |\{j \mid \mathcal{A}_{ij} = p \wedge e_j = q\}| \tag{1.1}$$

where $n_{pq}(i)$ is the number of runs in which the component c_i has been active during execution ($p = 1$) or not ($p = 0$), and in which the runs failed ($q = 1$) or passed ($q = 0$). For instance, $n_{11}(i)$ counts the number of times component c_i has been involved ($p = 1$) in failing executions ($q = 1$), whereas $n_{10}(i)$ counts the number of times component c_i has been involved in passing executions.

We then calculate similarity to the error vector by means of applying *fault predictors* (also referred to as *similarity coefficients*) to each component to produce a score quantifying how likely it is to be faulty. Components are then ranked according to such likelihood scores and reported to the user.

There is a large catalog of predictors to choose from [Lucia et al., 2014]. Table 1.1 lists a few fault predictors that are among the best performing ones in related work [Wong et al., 2016]. All of these fault predictors will provide a numerical score to components so that a ranked list of components for user inspection can be produced.

Table 1.1: Fault predictor formulas.

Predictor	Formula
D* [Wong et al., 2012; Wong et al., 2014]	$\frac{n_{11}(i)^*}{n_{01}(i)+n_{10}(i)}$
O [Abreu et al., 2009c; Naish et al., 2011]	$\begin{cases} -1 & \text{if } n_{01}(i) > 0 \\ n_{00}(i) & \text{otherwise} \end{cases}$
O^P [Naish et al., 2011]	$n_{11}(i) - \frac{n_{10}(i)}{n_{10}(i)+n_{00}(i)+1}$
Ochiai [Abreu et al., 2006]	$\frac{n_{11}(i)}{\sqrt{n_{11}(i)+n_{01}(i)}+\sqrt{n_{11}(i)+n_{10}(i)}}$
Tarantula [Jones and Harrold, 2005]	$\frac{\frac{n_{11}(i)}{n_{11}(i)+n_{01}(i)}}{\frac{n_{11}(i)}{n_{11}(i)+n_{01}(i)}+\frac{n_{10}(i)}{n_{10}(i)+n_{00}(i)}}$

A fault predictor named DStar (D*) was introduced by Wong et al. [2014] such that the likelihood of a component j being faulty is: (1) proportional to the number of failed tests that cover it, (2) inversely proportional to the number of successful tests that cover it, and (3) inversely proportional to the number of failed tests that do not cover it. Their intuition is that statement (1) and should carry a higher weight than statements (2) and (3). Therefore, DStar provides a $*$ parameter — where $* \geq 1$ — for changing the weight carried by $n_{11}(i)$ in the formula's numerator. In this paper, we use $* = 2$, since [Wong et al., 2012] shows that D^2 is more effective as a fault predictor than other similarity coefficients.

The O fault predictor is often called the *optimal* metric, but it assumes that there is only one fault affecting the system [Naish et al., 2011]. Given only one bug, then its $n_{01}(i)$ should always be zero, and therefore any component with a nonzero $n_{01}(i)$ is given the lowest score. Since $n_{11}(i) + n_{01}(i)$ equals the number of failing runs, and $n_{10}(i) + n_{00}(i)$ equals the number of passing runs, there is only one degree of freedom left, expressed by assigning $n_{00}(i)$ as the predictor's value, with the aim of minimizing $n_{10}(i)$. Assuming one bug in the system, O was proven to be optimal by Naish et al. [2011]. Note that this optimality does not necessarily mean that it performs always better than other predictors [Yoo et al., 2014].

As an attempt to relax the assumptions from O, Naish et al. also proposed the O^P fault predictor, which does not assign a negative score to every component j where

	n_{00}	n_{10}	n_{01}	n_{11}	D^2	O	O^P	Ochiai	Tarantula
c_1	0	1	1	2	2.0	-1	1.5	.58	.40
c_2	1	0	1	2	4.0	-1	2.0	.64	1.0
c_3	0	1	2	1	.33	-1	.50	.32	.25

Figure 1.3: Fault predictor outcomes for the example in Figure 1.2.

$n_{01}(i) > 0$ holds [Naish et al., 2011]. In contrast, O^P scores components first based on their involvement in failing transactions and second on their involvement on passing transactions.

Ochiai, used in the context of fault localization by Abreu et al. [2006], evaluates how similar a coverage matrix column $\mathcal{A}_j$ is from the error vector e—it is a proxy for calculating the *cosine similarity* between two N-dimensional vectors.

The Tarantula predictor was proposed by Jones and Harrold [2005] to assist fault localization using a visualization technique. The intuition behind this predictor is that components that are used often in failed executions, but seldom in passed executions, are more likely to be the root cause of observed failures.

Figure 1.3 shows component frequency values and fault predictor outcomes for the example in Figure 1.2. Note that the predictor O scores every component with its lowest score of -1. This is to be expected since no component in the spectrum from Figure 1.2 is active in all failing tests, and therefore the spectrum necessarily contains more than one fault. Component c_2 exhibits the highest score in all other predictors, which means these similarity-based *SFL* approaches will suggest that component to be inspected first.

1.2.2 Reasoning-based *SFL*

Reasoning-based *SFL*, introduced by Abreu et al. [2009c], relies on an approach that leverages a Bayesian reasoning framework to diagnose the system, which is able to also diagnose multiple, intermittent faults. The two main steps of the reasoning-based methodology are candidate generation and candidate ranking:

Candidate Generation The first step in reasoning-based *SFL* is to generate a set $\mathcal{D} = \{d_1, d_2, \cdots, d_k\}$ of diagnosis candidates. Each diagnosis candidate d_k is a subset of $\mathcal{C}$, and d_k is said to be valid if every failed transaction involved at least one component from d_k. A candidate d_k is *minimal* if no other valid candidate d' is contained in d_k. We are only interested in minimal candidates, as they can subsume others of higher cardinality. Heuristic approaches to finding these minimal candidates, which is an instance of the **Minimal Hitting Set (*MHS*)** problem, thus NP-hard, include STACCATO [Abreu and van Gemund, 2009], SAFARI [Feldman et al., 2008] and MHS^2 [Cardoso and Abreu, 2013b].

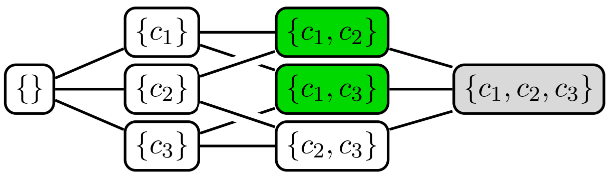


Figure 1.4: *Hasse* diagram of diagnostic candidates.

All possible (but not necessarily *valid*) candidates of the example spectrum from Figure 1.2 are depicted in the *Hasse* diagram from Figure 1.4. Colorless nodes represent invalid candidates—ones whose components cannot explain every failing transaction. Green nodes depict *minimal* valid candidates. Gray nodes depict valid *non-minimal* candidates which can be subsumed by at least one minimal candidate. Therefore, the collection of minimal valid diagnostic candidates that can explain the erroneous behavior is:

- $d_1 = \{c_1, c_2\}$
- $d_2 = \{c_1, c_3\}$

Candidate Ranking For each candidate d_k, their fault probability is calculated using the Naïve Bayes rule

$$\Pr(d_k \mid (\mathcal{A}, e)) = \Pr(d_k) \cdot \prod_{j \in 1..N} \frac{\Pr((\mathcal{A}_j, e_j) \mid d_k)}{\Pr(\mathcal{A}_j)} \tag{1.2}$$

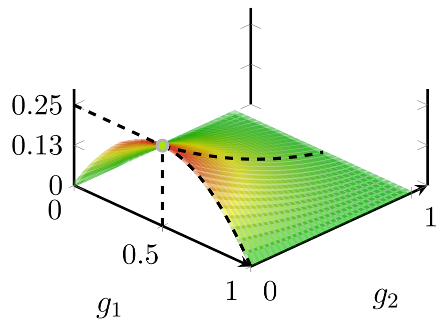
Let A_j be short for $\{\mathcal{A}_{ij} | i \in 1..M\}$ — *i.e.*, the jth column of matrix $\mathcal{A}$, represented by a set encoding all component involvements in test t_j. The denominator $\Pr(\mathcal{A}_j)$ is a normalizing term that is identical for all candidates and is not considered for ranking purposes.

In order to define $\Pr(d_k)$, let p_i denote the prior probability[4] that a component c_i is at fault. The prior probability for a candidate d_k is given by

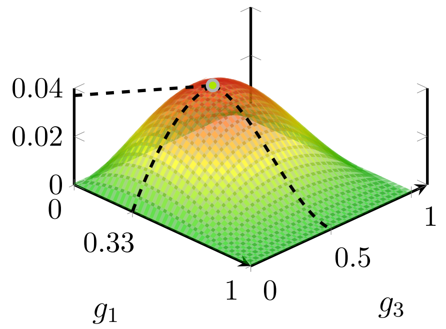
$$\Pr(d_k) = \prod_{i \in d_k} p_i \cdot \prod_{i \in C \setminus d_k} (1 - p_i) \tag{1.3}$$

$\Pr(d_k)$ estimates the probability that a candidate, without further evidence, is responsible for erroneous behavior.

[4]Component prior probabilities depend on the chosen granularity. For instance, if components are statements, one can approximate p_j as $1/1000$, *i.e.*, 1 fault for each 1000 lines of code [Carey et al., 1999].



(a) $\Pr(d_1 \mid (\mathcal{A}, e))$



(b) $\Pr(d_2 \mid (\mathcal{A}, e))$

Figure 1.5: Maximum Likelihood Estimation of candidate probability functions.

$\Pr((\mathcal{A}_j, e_j) \mid d_k)$ is used to bias the prior probability taking observations into account. Let g_i (referred to as component goodness) denote the probability that a component c_i performs nominally

$$\Pr((\mathcal{A}_j, e_j) \mid d_k) = \begin{cases} \prod\limits_{j \in (d_k \cap \mathcal{A}b_i)} g_j & \text{if } e_j = 0 \\ 1 - \prod\limits_{j \in (d_k \cap \mathcal{A}b_i)} g_j & \text{otherwise} \end{cases} \tag{1.4}$$

In cases where values for g_i are not available they can be estimated by maximizing $\Pr((\mathcal{A}, e) \mid d_k)$ — *i.e.*, **Maximum Likelihood Estimation (*MLE*)** for the Naïve Bayes classifier — under parameters $\{g_i \mid i \in d_k\}$ [Abreu et al., 2009b]. This work uses *MLE* to estimate component goodness.

If we consider our example from Figure 1.2, the probabilities for both candidates are

$$\Pr(d_1 \mid (\mathcal{A}, e)) = \overbrace{\left(\frac{999}{1000} \cdot \frac{999}{1000} \cdot \frac{1}{1000}\right)}^{\Pr(d_1)} \times \overbrace{\underbrace{(1 - g_1 \cdot g_2)}_{t_1} \cdot \underbrace{(1 - g_2)}_{t_2} \cdot \underbrace{(1 - g_1)}_{t_3} \cdot \underbrace{g_1}_{t_4}}^{\Pr((\mathcal{A},e)|d_1)} \tag{1.5}$$

$$\Pr(d_2 \mid (\mathcal{A}, e)) = \overbrace{\left(\frac{999}{1000} \cdot \frac{1}{1000} \cdot \frac{999}{1000}\right)}^{\Pr(d_2)} \times \overbrace{\underbrace{(1 - g_1)}_{t_1} \cdot \underbrace{(1 - g_3)}_{t_2} \cdot \underbrace{(1 - g_1)}_{t_3} \cdot \underbrace{g_1 \cdot g_3}_{t_4}}^{\Pr((\mathcal{A},e)|d_2)} \tag{1.6}$$

By performing an *MLE* for both functions it follows that Equation (1.5) is maximized for $g_1 = 0.50$ and $g_2 = 0.00$. Equation (1.6) is maximized for $g_1 = 0.33$ and $g_3 = 0.50$ (see Figure 1.5). Applying the goodness values to both expressions, it follows that $\Pr(d_1 \mid (\mathcal{A}, e)) = 1.9 \times 10^{-9}$ and $\Pr(d_2 \mid (\mathcal{A}, e)) = 4.0 \times 10^{-10}$. It is customary to normalize fault probabilities over the set of candidates under consideration, producing: $\Pr(d_1 \mid (\mathcal{A}, e)) = 0.83$ and $\Pr(d_2 \mid (\mathcal{A}, e)) = 0.17$, entailing the ranking $\langle d_1, d_2 \rangle$.

1.2.3 Measuring Diagnostic Accuracy

To measure the accuracy of fault-localization approaches, the cost of diagnosis C_d metric is often used [Abreu et al., 2009c; Campos et al., 2013; Steimann et al., 2013; Wong et al., 2016]. Given the true fault d_t and the *SFL* diagnostic report $\mathcal{R} = \{\text{likelihood}(d) | d \in \mathcal{D}\}$, where likelihood$(d)$ either denotes the fault predictor score (in the case of similarity-based *SFL*[5]) or fault probability (in reasoning-based variants), then C_d can be computed as

$$C_d = |\{l > \text{likelihood}(d_t) | l \in \mathcal{R}\}| + \frac{|\{l = \text{likelihood}(d_t) | l \in \mathcal{R}\}|}{2} \tag{1.7}$$

C_d measures the average number of diagnostic candidates that need to be inspected until the real fault is reached, given that the candidates are being inspected by descending order of fault likelihood, as computed by *SFL*. A value of 0 for C_d indicates an ideal diagnostic report where the faulty candidate is at the top of the ranking and thus no spurious code inspections occur. Another common metric is Wasted Effort (or merely Effort), that normalizes C_d over the total number of diagnostic candidates, so that the metric ranges from 0 (optimal value—no developer time wasted chasing wrong leads) to 1 (worst value—in which all diagnostic candidates will be inspected until the fault is reached) in all cases.

1.3 Research Goals

SFL was shown to be a lightweight, efficient technique that help developers pinpoint bugs in software. Its language-agnostic nature makes it exceptionally versatile and extensible. *SFL* was also proven effective when coupled with more computationally taxing diagnostic techniques—such as model-based diagnosis [Abreu et al., 2009a] and dynamic slicing [Hofer and Wotawa, 2012]—, as the spectrum-based abstraction allows for a thorough pruning of their search spaces.

Despite the aforementioned advantages that *SFL* tools provide, we are unable to find many accounts of successful transitions of this technology into the software industry at large. This is motivated largely by the issues pointed out by Parnin and Orso in their 2011 study of automated debugging techniques [Parnin and Orso, 2011], which were corroborated by Steimann et al. [2013] and Pearson et al. [2017]. Several problems were outlined in the aforementioned studies, not the least important of which were (1) the reliance on a *thorough test-suite*, and (2) the fact there is a

[5]Diagnostic candidates in similarity-based *SFL* are single components (*i.e.*, $\mathcal{D} = \mathcal{C}$).

considerable interest drop-off after developers inspect a small number of diagnostic candidates.

Without a thorough test-suite, the applicability and accuracy of *SFL* can be significantly hindered, since the crux of the technique is that components which are frequently covered by failing tests are more likely to be faulty than those more frequently covered by passing tests. As will be shown in Section 2.2, having a test-suite that satisfies an adequacy criterion (such as full statement coverage) might still not yield a *diagnosable* system, due to, *e.g.*, poor fault isolation. Therefore, in order to ensure the diagnosability of the system, one should take into account the fact that test cases created in the testing phase of the development cycle will be used by downstream techniques such as *SFL*. It is thus crucial to account for the test-suite's propensity for diagnosis when developing tests in a measurable, actionable fashion. Similarly, due to the abstract nature of program spectra, fault comprehension—*i.e.*, the ability to identify a faulty component as such when inspecting it—may be hindered. This aspect is oft overlooked by diagnostic performance metrics, such as the ones presented in Section 1.2.3. Therefore, novel ways to provide more contextual information detailing why candidates are considered suspicious should be embedded in the diagnostic analysis and subsequent diagnostic report.

The abstract nature of spectrum-based analyses also lends itself useful in other tasks, as evidenced by work on the diagnosis of spreadsheets [Abreu et al., 2014; Hofer et al., 2015]. A particular software development task that shares many similarities with debugging is feature localization—*i.e.*, the task of locating where a certain unit of functionality is implemented in the source-code. This task is important not only for novice engineers to understand the inner workings of software systems, but also as the first step in any software maintenance or evolution scenario.

In this thesis, we aim to consider spectrum-based diagnostic analyses and apply them throughout the development life cycle in a more *holistic* manner, with the intent of not only mitigating the issues described above but also producing *higher-quality* software that is more *diagnosable* and more *maintainable*. In particular, the main question this thesis addresses is the following:

Can we improve the usefulness and effectiveness of spectrum-based analyses throughout the software development life cycle?

Below, we outline four research goals that will help us answer the question posed above.

1.3.1 Diagnosability Assessment

SFL, being a dynamic analysis technique, is heavily dependent on a *good* test suite that is capable of not only *detecting* errors, but also of *isolating* them. Let us define the latter property as *diagnosability*:

Definition 6 (Diagnosability) *The ability of a test suite to effectively and accurately locate faults when errors arise.*

In order to assess if test suites are diagnosable, we need a way to *measure* diagnosability. Previous test-suite diagnosability research has proposed measurements to assess diagnostic efficiency of spectrum-based fault localization techniques. González-Sanchez et al. [2011b] used activity matrix density (ρ) as a measure for diagnosability:

$$\rho = \frac{\sum_{i,j} \mathcal{A}_{ij}}{N \times M} \tag{1.8}$$

The intuition was to find an optimal matrix density such that every transaction observed reduces the entropy of the diagnostic report set $\mathcal{R}$. It has been previously demonstrated that the information gain can be modeled as:

$$\begin{aligned} IG(t_g) = & - \Pr(e_g = 1) \cdot \log_2(\Pr(e_g = 1)) \\ & - \Pr(e_g = 0) \cdot \log_2(\Pr(e_g = 0)) \end{aligned} \tag{1.9}$$

where $\Pr(e_g = 1)$ is the probability of observing an error in transaction t_g, conversely $\Pr(e_g = 0)$ is the probability of observing nominal behavior. Optimal information gain ($IG(t_g) = 1$) is achieved when $\Pr(e_g = 1) = \Pr(e_g = 0) = 0.5$. With the assumption that transaction activity is normally distributed, then it follows that a transaction's average component activation rate equals the overall matrix density. Thus, it can be said that $\Pr(e_g = 1) = \rho$, yielding $\rho = 0.5$ as the ideal value for diagnosis using *SFL* approaches [González-Sanchez et al., 2011b].

Another diagnosability measurement was proposed by Baudry et al. [2006], which tracks the number of *dynamic basic blocks* in a system. Dynamic basic blocks, which other authors also call *ambiguity groups* [González-Sanchez et al., 2011a], correspond to sets of components that exhibit the same involvement pattern across the entire test-suite. For diagnosing a system, the more ambiguity groups there are, the less accurate the diagnostic report can be, because one cannot distinguish among components in a given ambiguity group, as they all show the same involvement pattern across every transaction. The metric, which we call uniqueness, will therefore ensure that the test-suite is able to break as many ambiguity groups as possible. To compute the metric, consider that an activity matrix $\mathcal{A}$ decomposes the system into a partition $G = \{g_1, g_2, \cdots, g_L\}$ of subsets of components with identical rows in $\mathcal{A}$ such that $\forall a, b \in g, \forall j \in \mathcal{T} : \mathcal{A}_{aj} = \mathcal{A}_{bj}$. Then, measuring the uniqueness $\mathcal{U}$ of a system can be done by

$$\mathcal{U} = \frac{|G|}{M} \tag{1.10}$$

When $\mathcal{U} = 1/M$, all components belong to the same ambiguity group. When $\mathcal{U} = 1$, all components exhibit different coverage patterns and consequently can be uniquely identified.

Unfortunately, the aforementioned diagnosability metrics rely on impractical assumptions that are unlikely to happen in the real world. The approach by Baudry et al. assumes all faults can trigger by exercising a single component. The density approach assumes that all tests exercise distinct code paths and therefore would produce different coverage patterns. In practice, it is common for different tests to cover the same code path. If one does not account for test diversity, it is possible to skew the test-coverage matrix to have a (supposedly) optimal density by repeating similar test cases. These assumptions will be further examined in Section 2.3.

The theoretically ideal activity matrix for *SFL* diagnosis is depicted in Figure 1.6. This ideal matrix is one that contains every combination of component activations since it follows that every possible fault candidate in the system is exercised.

$$
\begin{array}{c|ccccc}
 & t_1 & t_2 & t_3 & \cdots & t_{2^M-1} \\
\hline
c_1 & 1 & 0 & 0 & \cdots & 0 \\
c_2 & 0 & 1 & 0 & \cdots & 0 \\
c_3 & 0 & 0 & 1 & \cdots & 0 \\
c_4 & 0 & 0 & 0 & \cdots & 1 \\
c_5 & 1 & 1 & 0 & \cdots & 0 \\
\vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\
c_M & 1 & 1 & 1 & \cdots & 1
\end{array}
$$

Figure 1.6: Ideal activity matrix for a system with M components.

However, this ideal matrix would also be impractical to achieve, as one would need to create $2^M - 1$ tests. Nevertheless, this optimal scenario can help elicit a set of essential properties that activity matrices need to exhibit for accurate spectrum-based fault localization. Therefore, our first research goal is:

Research Goal 1

Can we create a near-optimal metric to assess *SFL* diagnosability of test suites while avoiding the assumptions held in previous work?

1.3.2 Cardinality of Fixes

There is a considerable amount of research focusing on similarity-based *SFL* when compared to reasoning-based *SFL*, even though the latter was shown to be more accurate and shown to be able to diagnose more than one fault at a time [Abreu et al., 2009c]. But this gain in accuracy naturally comes at a cost, given that both the *MHS* computation in the candidate generation step and the *MLE* computation in the candidate ranking step of the approach are computationally expensive tasks, especially when compared to the simplicity of most fault predictors used in similarity-based approach. This leads us to question whether the use of the (significantly faster) similarity-based approaches is preferred over (slower) more intricate approaches, in the event that single-faults are more prevalent in practice. Let us define *fix cardinalities*:

Definition 7 (Single-Fault Fix) *A fix that eliminates one fault from the system.*

Definition 8 (Multiple-Fault Fix) *A fix that eliminates more than one fault from the system.*

It remains to be seen is how often such multiple-fix scenarios—that would benefit from reasoning-based approaches to help diagnose multiple faults— actually happen in practice. Our hypothesis is that, more often than not, programmers detect and fix *one bug at a time* during development. This would mean that, most often, developers are faced with single-faulted scenarios, so the use of (faster) fault predictors would be justified. Note that this hypothesis does not necessarily state that systems are at most single-faulted at any given time, but rather that faults are mostly detected in isolation. Our research goal is then:

Research Goal 2

What is the prevalence of single-fault fixes versus multiple-fault ones, and what is their impact in similarity-based *SFL*?

1.3.3 Augmenting Spectra

SFL faces several issues preventing it from widespread adoption and use. Not the least of which is the lack of contextual information, essential for understanding *why* diagnostic candidates are considered suspicious. This has been pointed out recurrently in the fault localization literature [Parnin and Orso, 2011; Steimann et al., 2013; Pearson et al., 2017]. As *SFL* reasons about failures at the coverage level, it only has access to whether a software component was involved or not in each test case. While this enables a language-independent, lightweight analysis, the necessary abstraction can impose a tradeoff both in accuracy and comprehension. Although there have been efforts to incorporate more data into the diagnostic process—by modeling component behavior that considers the system's state and previous diagnoses [Cardoso and Abreu, 2013a]; or by leveraging prediction models trained from version control and issue tracking data [Elmishali et al., 2016]—they were focused on *conditioning* the fault probability of existing diagnostic candidates, increasing accuracy but not necessarily increasing the ability to *comprehend* the diagnostic report.

Aside from *comprehension*, and because *SFL* only reasons about component involvement and similarity to failing tests, omission errors—errors that result from missing code [Basili and Perricone, 1984], such as forgetting to check whether a divisor is

`def purchase_item(item_id, price, quantity, wallet):`	t_1	t_2	t_3
`  total = price * quantity_discount(item, quantity)`	●	●	●
`  balance = balances.get(wallet)`	●	●	●
`  if balance - total >= 0:`	●	●	●
`    balance = balance - total`		●	●
`    update_balance(wallet, balance)`		●	●
`    update_stock_quantity(item, quantity)`		●	●
`    return True`		●	●
`  return False`	●		
`def quantity_discount(item, quantity):`			
`  return quantity * price_rates.get(item)(quantity)`	●	●	●
`def update_balance(wallet, balance):`			
`  balances.update(wallet, balance)`		●	●
`def update_stock_quantity(item, quantity):`			
`  stock = quantities.get(item)`		●	●
`  quantities.update(item, stock - quantity)`		●	●
Test outcome (e):	✓	✓	✗

Figure 1.7: Code snippet and respective spectrum exhibiting coincidental correctness.

equal to zero—also become difficult to diagnose [Xu et al., 2011; Xie et al., 2013]. The abstract nature of the spectra that is fed into current *SFL* frameworks also facilitates the occurrence of *coincidental correctness*:

Definition 9 (Coincidental correctness) *The event when no failure is detected, even though a fault has been executed [Richardson and Thompson, 1993; X. Wang et al., 2009].*

Depending on the component granularity selected for the analysis, coincidental correctness can happen at a frequent rate. In particular, when two tests share the same coverage path, but produce different outcomes, it becomes significantly harder to distinguish them without further contextual information. Coincidental correctness can potentially lead to exonerating real faults from inspection as they are observed to behave nominally.

Consider the code snippet from Figure 1.7. The code implements a (simplified) item purchasing system. Function `purchase_item` takes as parameters an item identifier along with its unit price and quantity, and an identifier for the customer's virtual wallet. Test t_2 performs a purchase with a wallet that contains enough funds to successfully finish the transaction. The predicate `balance - total >= 0` yields `True`, and almost all statements in the snippet are executed. Conversely, test t_3 attempts to purchase an item with a negative price. Although unusual, one can think of several scenarios where an item's price might become negative, such as after applying a rebate or coupon code. However, even if this scenario happens, a user's wallet should never increase its balance after a transaction, which is what happens in t_3, yielding a test failure. This is an *error of omission* in the code. Function `purchase_item` should check if `total` is negative and change its value to zero if that is the case. Tests t_2 and t_3 both exhibit the same coverage pattern, but the former has a passing outcome while the latter fails. This is an example of *coincidental correctness*. One way to remove the ambiguity between the two tests would be, for instance, by inspecting the sign of the `price` parameter, as a contextual way to distinguish between the two tests. We argue that the issues described above can be prevented, or at least attenuated, if we supplement the *SFL* framework with more contextual information about the system under analysis when diagnosing. Our third research goal is thus:

Research Goal 3

Can we augment spectra with qualitative, contextual information to improve *SFL* diagnoses?

1.3.4 Feature Localization

Let us define a software *feature* as follows:

Definition 10 (Feature) *A feature is the source-code portion that implements a certain functionality. It can encompass one or more components.*

This definition is closely related to the concept of feature in the domain of software product-line research: *"a feature is a unit of functionality of a software system that satisfies a requirement, represents a design decision, and provides a potential configuration option"* [Apel et al., 2013].

Typically in software maintenance or evolution scenarios, developers need to *find* features in the source code to further develop them or possibly revise them. Also, it is not uncommon for developers to spend 60% to 80% of their time in such feature location and comprehension tasks [Tiarks, 2011]. We argue that the process of finding features is not too dissimilar with what happens when diagnosing faults, as in both cases developers need to pinpoint the root cause of a particular software *behavior*. Therefore, we hypothesize that the feature localization task can be framed as a software diagnosis effort and consequently leverage the approaches and tooling that exist in the fault localization domain.

Research Goal 4

Can we leverage *SFL* for feature detection?

1.4 Origin of Chapters

All chapters in this thesis have either been published in peer-reviewed venues or are currently under review. All publications have been co-authored with Rui Abreu. Publication of Chapter 2 has also been co-authored with Arie van Deursen. Publication of Chapter 3 has been co-authored with Marcelo d'Amorim.

Chapter 2 is based on work submitted to the *IEEE Transactions on Software Engineering*, which is currently under review. An earlier version of this work was published in the *Proceedings of the International Conference on Software Engineering (ICSE'17)* [Perez et al., 2017b].

Chapter 3 is based on the work from [Perez et al., 2017a], published in the *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST'17)*.

Chapter 4 is based on work from [Perez and Abreu, 2018a], to be published in the *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI'18)*. A preliminary version of this work will appear as a poster in the *International Conference on Software Engineering (ICSE'18)* [Perez and Abreu, 2018b].

Chapter 5 is based on work from [Perez and Abreu, 2016], which was published in the *Journal of Software: Evolution and Process*, 2016. An earlier version of this work appeared in the *Proceedings of the International Conference on Program Comprehension (ICPC'14)* [Perez and Abreu, 2014].

1.5 Thesis Outline

Figure 1.8 depicts the outline for the rest of the thesis. We apply a more holistic view on the use of spectrum-based diagnostic analyses throughout the software development life cycle (denoted as the root of the diagram). In particular, we:

- Propose a diagnosability measurement evaluating the effectiveness of spectrum-based approaches for a given test-suite, to be used in the testing phase—tackling research goal 1 in Chapter 2;
- Study how faults are actually fixes in practice during the fault fixing step of the debugging phase—tackling research goal 2 in Chapter 3;
- Improve the fault localization step of the debugging phase by augmenting spectra with contextual, qualitative runtime descriptions—tackling research goal 3 in Chapter 4;
- Apply spectrum-based diagnostic analyses to the task of locating features during the maintenance phase—tackling research goal 4 in Chapter 5.

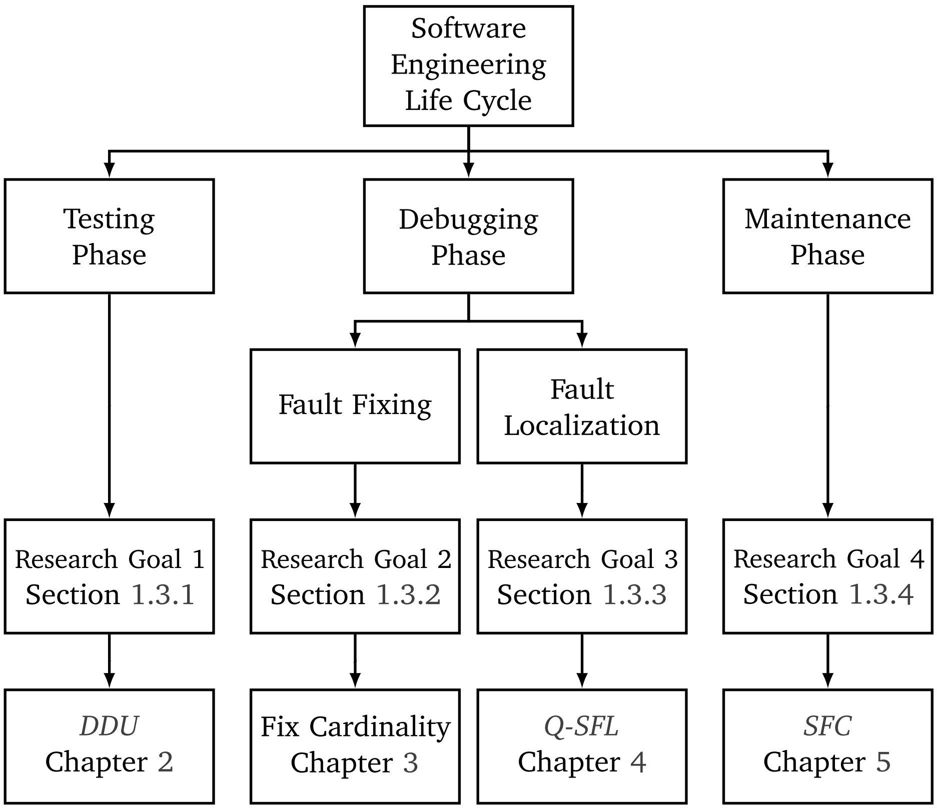

Software Engineering Life Cycle
Testing Phase
Debugging Phase
Maintenance Phase
Fault Fixing
Fault Localization
Research Goal 1 Section 1.3.1
Research Goal 2 Section 1.3.2
Research Goal 3 Section 1.3.3
Research Goal 4 Section 1.3.4
DDU Chapter 2
Fix Cardinality Chapter 3
Q-SFL Chapter 4
SFC Chapter 5

Figure 1.8: Thesis Outline.

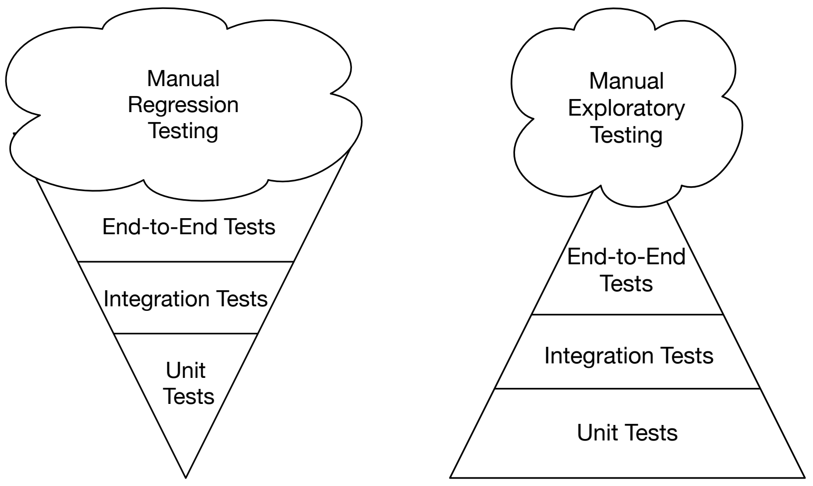
A Diagnosability Measurement 2

A Test-Suite Diagnosability Metric for Spectrum-based Fault Localization Approaches
Alexandre Perez, Rui Abreu, and Arie van Deursen
In: Proceedings of the 39th International Conference on Software Engineering, ICSE'17, Buenos Aires, Argentina, May 20-28, pp. 654–664, 2017.

Abstract *Current metrics for assessing the adequacy of a test-suite plainly focus on the number of components (be it lines, branches, paths) covered by the suite, but do not explicitly check how the tests actually exercise these components and whether they provide enough information so that spectrum-based fault localization techniques can perform accurate fault isolation. We propose a metric, called DDU, aimed at complementing adequacy measurements by quantifying a test-suite's diagnosability,* i.e., *the effectiveness of applying spectrum-based fault localization to pinpoint faults in the code in the event of test failures. Our aim is to increase the thoroughness of test-suites, so they are not only regarded as error detection mechanisms but also as effective diagnostic aids that help widely-used fault-localization techniques to accurately pinpoint the location of bugs in the system. We have performed a topology-based simulation of thousands of spectra and have found that DDU can not only effectively establish an upper bound on the effort to diagnose faults, but also lower bound the fault detection rate. Furthermore, our empirical experiments using the* DEFECTS4J *dataset show that optimizing a test suite with respect to DDU yields a 34% gain in spectrum-based fault localization report accuracy when compared to the standard branch-coverage metric.*

2.1 Introduction

Current test quality metrics quantitatively describe how close a test-suite is to thoroughly exercising a system according to an adequacy criterion. Such criteria describe what characteristics of a program must be exercised. Examples of current metrics include branch and path coverage [J. C. Miller and Maloney, 1963], modified decision/condition coverage [Chilenski and S. P. Miller, 1994], and mutation coverage [Budd, 1980]. According to Zhu et al. [1997], such measurements can act as *generators*, meaning that they provide an intuition on *what* components to exercise to improve the suite. However, this *generator* property does not provide any relevant, actionable information on *how* to test those components. These adequacy measurements abstract away the execution information of single test executions to favor an overall assessment of the suite, and are therefore oblivious to anti-patterns like the



(a) Anti-Pattern. Prevalence of manual and end-to-end testing.

(b) Best-practice. Prevalence of unit tests.

Figure 2.1: Alister Scott's *ice-cream cone* software testing anti-pattern.

ice-cream cone[1] (depicted in Figure 2.1). The anti-pattern states that often times the vast majority of tests is written at the system level (or even executed manually in an *ad-hoc* manner), with very few tests written at the unit granularity level (Figure 2.1a), whereas in most software development scenarios we should expect the opposite as a best-practice (Figure 2.1b). Even though high-coverage test-suites (such as end-to-end tests) may detect errors in the system, it is not guaranteed that inspecting such failing tests will yield a straightforward explanation for the cause of the observed failures, since fault isolation is not a typically primary concern. Our hypothesis is that a complementing metric that takes into account per-test execution information can provide further insight about the overall quality of a test-suite. This way, if a regression happens, one would have a test suite that is not only effective at *detecting* faults, but also aids spectrum-based techniques to efficiently *pinpoint* them among the code.

As outlined in Section 1.3.1, previous test-suite diagnosability research has proposed measurements to assess diagnostic efficiency of *SFL* techniques. One measurement uses the density (ρ) of the activity matrix. González-Sanchez et al. [2011b] showed that when spectrum density approaches an optimal value, the effectiveness of spectrum-based approaches is maximal. Another approach, by Baudry et al. [2006], proposes a *test for diagnosis* criterion that attempts to reduce the size of *dynamic basic blocks* to improve fault localization accuracy. The aforementioned metrics rely on impractical assumptions that are unlikely to happen in the real world—namely by presuming faults trigger by exercising single components, or by presuming all tests written by programmers will invariably exercise a different path through the code

[1]Ice-cream cone software testing anti-pattern mentioned in Alister Scott's blog: `http://goo.gl/bhXOrN`.

and consequently produce different coverage patterns. If one does not explicitly account for test diversity, it is possible to skew the test-coverage matrix to have a (supposedly) optimal density by repeating similar test cases, as will be demonstrated in Section 2.3.2.

We depict in Section 1.3.1 (Figure 1.6) the optimal coverage matrix for achieving accurate spectrum-based fault localization. In this optimal scenario, the test-suite contains a test case exercising every possible combination of components in the system, so that not only single- or multiple-faults can be pinpointed but also allows for scenarios which require simultaneous activations of components for the fault to manifest. Such a matrix is reached when its entropy is maximal. This is the theoretically optimal scenario. However, the entropy-maximization approach is intractable due to the sheer number of test cases required to exercise every combination of components in any real-world system.

Nevertheless, the entropy-optimal scenario helps elicit a set of properties coverage matrices need to exhibit for accurate spectrum-based fault localization. We leverage these properties in our proposed metric, coined **Density-Diversity-Uniqueness (*DDU*)**. This metric addresses the related work assumptions detailed above, while still ensuring tractability, by combining into a single measurement the three key properties spectra ought to have for practical and efficient diagnosability: (1) density (ρ), ensuring components are frequently involved in tests; (2) test diversity ($\mathcal{G}$), ensuring components are tested in diverse combinations; and (3) uniqueness ($\mathcal{U}$), favoring spectra with less ambiguity among components and providing a notion of component distinguishability. The metric addresses the quality of information gained from the test-suite, should a program require fault-localization activities, and is intended as a complement to adequacy measurements such as branch-coverage.

To measure the effectiveness of the proposed metric, we perform theoretical and empirical evaluations. The theoretical evaluation simulates a vast breadth of software systems and test suite compositions so that the range of *DDU* values can be effectively generated and analyzed in a holistic manner. Our simulation is built upon a tree-based representation of system structures—which we call topologies—that are randomly generated following phylogenetic processes. Topologies then guide the generation of multiple spectra, which are then fault-injected and diagnosed. This theoretical analysis reveals that *DDU* can effectively predict an upper-bound on the effort required to diagnose. We also empirically evaluate *DDU* by generating test suites for real-world faulty software projects. Test generation, facilitated by the EVOSUITE tool, is guided to optimize test suites regarding a specific metric, and oracles are generated from correct project versions. The first empirical evaluation shows that generating tests that optimize *DDU* produces test-suites that require less diagnostic effort to find faults compared to the state-of-the-art of diagnosability metrics, such as density. The second empirical evaluation generates test-suites for a wide range of subjects in the DEFECTS4J collection. We provide empirical evidence that optimizing a suite regarding *DDU* yields an increase of 34% in diagnostic accuracy when compared to test-suites that only consider branch-coverage (*i.e.*, a test adequacy measurement) as the optimization criterion.

This chapter's contributions are:

- We elicit from the optimal entropy scenario three key properties matrices ought to exhibit to preserve high diagnostic accuracy: density, diversity and uniqueness.
- *DDU*, a new metric based on the aforementioned properties to assess a test-suite's diagnostic ability to pinpoint a fault in the system using spectrum-based techniques. The metric complements adequacy measurements such as branch-coverage.
- A large-scale theoretical evaluation of *DDU* through a topology-based program spectra simulation able to generate and analyze a vast breadth of qualitatively distinct faulty spectra.
- Empirical evidence that *DDU* is more accurate at assessing diagnostic ability than the state-of-the-art.
- Empirical evidence that optimizing a test-suite with respect to *DDU* yields a 34% gain in diagnostic efficiency when compared to similarly adequate suites.

2.2 Motivation

We present two code snippets along with runtime information of several test cases as a motivational example demonstrating the need for a new metric that accurately describes the diagnostic ability of a test-suite[2].

The first example, depicted in Figure 2.2a, shows a snippet of code from a sensor array capable of measuring distance to the ground both when submerged and airborne. The purpose of `groundAltitude` is to measure distance to the ground using the internal altitude sensor (`ALT`) and the ground elevation sensor (`GND`). This method has a bug: it will produce negative values if `ALT` is greater than `GND`. The line should then read `return sub(ALT, GND)`. Test t_1 does indeed detect the error in the system. But the problem is that no other test also exercises the branches followed by t_1 to exonerate them from suspicion. This results in the developer having to manually inspect all components that do not appear in passing tests. Six lines out of a total of 12 will have to be inspected, corresponding to nearly 50% of the total code in the snippet. In this small example, it is feasible to inspect all components, but component inspection slices can grow to fairly large numbers in a real world

[2]We use line of code as the component granularity throughout the motivation section.

	t_1	t_2	t_3	t_4
`def groundDistance():`	●	●		●
`  if underwater():`	●	●		●
`    return surfaceDistance()`		●		●
`  else:`	●			
`    return groundAltitude()`	●			
`def groundAltitude():`	●		●	●
`  if landed():`	●		●	●
`    return 0`			●	●
`  else:`	●			
`    return sub(GND, ALT)`	●			
`def sub (a,b):`	●			
`  return a - b`	●			
Test outcome (e):	✗	✓	✓	✓

(a) Per-test coverage of a single-faulted system.

	t_1	t_2	t_3	t_4
`def descend(increment):`	●	●		●
`  if landed():`	●	●		●
`    return Status.STOPPED`	●			●
`  else:`		●		
`    descendMeters(increment)`		●		
`    return Status.DESCENDING`		●		
`def ascend(increment):`	●		●	
`  if landed():`	●		●	
`    liftoff()`	●			
`    return Status.LIFTOFF`	●			
`  else:`			●	
`    ascendFeet(increment)`			●	
`    return Status.ASCENDING`			●	
Test outcome (e):	✓	✓	✓	✓

(b) Per-test coverage of a multiple-faulted system.

Figure 2.2: Motivational code snippets with respective program spectrum.

scenario. So, even though this test-suite has 100% branch-coverage, it does not provide many diagnostic clues.

The second example, depicted in Figure 2.2b, contains a snippet of code for controlling the ascent and descent of a drone. The `descend` method uses meters to quantify the amount of descent, while the `ascend` method uses feet. Assuming there is no explicit check for altitude available, testing these methods independently will not reveal the failure. In fact, only a test that covers both methods' else branches may reveal it if, for instance, there is an unexpected liftoff after a descent. Even though we have reached 100% branch coverage, this test-suite has not managed to expose the fault in the code. Also note that even satisfying a stronger coverage criterion like the modified condition/decision coverage or even a stronger intra-procedural analysis will not expose the fault. To expose the fault in this example one would need to exercise combinations of decisions from different methods.

2.3 The *DDU* Diagnosability Metric

As detailed in Section 1.3.1, the ideal activity matrix is one that contains every combination of component activations—depicted in Figure 1.6. A metric that accurately captures this exhaustiveness is entropy—the measure of uncertainty in a random variable. Shannon Entropy [Shannon, 2001] is given by

$$H(X) = -\sum_{i} P(x_i) \cdot \log_2(P(x_i)) \tag{2.1}$$

in this context, X is the set of unique transaction activities in the spectrum matrix. $P(x_i)$ is the probability of selecting a transaction $t \in \mathcal{T}$ and it having the same activity pattern as x_i. When $H(X)$ is maximal, it means that all possible transactions are present in the spectrum. For a system with M components, maximum entropy is M *shannons* (*i.e.*, number of bits required to represent the test suite). Therefore, we can normalize it to $^{H(X)}/_{M}$. Matrices with a normalized entropy of 1.0 would, then, be able to efficiently diagnose any fault (single or multiple) provided that the error detection oracles that classify transactions as faulty are sufficiently accurate. The main downside of using entropy as a measure of diagnosability is that one would need $2^M - 1$ tests to achieve this ideal spectrum (and thus a normalized entropy of 1.0). In practice, some transaction activities are impossible to be generated, either due to the system's topology or due to the existence of ambiguity groups: a set of components that always exhibit the same activity pattern[3].

Our goal, then, is to capture several structural properties of activity matrices that make them ideal for diagnosing, while avoiding the combinatorial explosion of the optimal entropy approach. We start by considering activity matrix density as the basis for our approach, and then propose the diversity and uniqueness enhancements so that the impractical assumptions of the base approach can be lifted.

2.3.1 Density

The ρ metric captures the density of a system. Its ideal value for minimizing the *diagnostic report entropy* is 0.5, as shown in the work of González-Sanchez et al. [2011b]. It is also straightforward to show the optimality of the value of 0.5 for the density measurement by induction, as depicted in Figure 2.3. Suppose that we have an activity matrix $\mathcal{A}'$, which is optimal for diagnosis. Suppose also that we want to add a new component c' to our system. To preserve optimality, we would need

[3]An example of an ambiguity group is the set of statements in a basic block.

to repeat the optimal sub-matrix $\mathcal{A}'$ both when c' is active and when it is inactive. Therefore, the involvement rate of the newly created component c' would be 0.5.

$$\begin{array}{c|ccc|ccc}
 & t_1 & \cdots & t_j & t'_1 & \cdots & t'_j \\
\hline
c_1 & & & & & & \\
\vdots & & \mathcal{A}' & & & \mathcal{A}' & \\
c_i & & & & & & \\
\hline
c' & 0 & 0 & 0 & 1 & 1 & 1
\end{array}$$

Figure 2.3: Depiction of the optimal density proof.

Note that in the case of *dependent faults*—ones where multiple simultaneous components must be involved for the fault to trigger —the optimal value depends on the fault cardinality. Suppose that a system contains N_f dependent faults. The total number of fault candidates can then be expressed by the binomial coefficient $\binom{C}{N_f}$. If the system's coverage matrix density is ρ, tests that exercise it cover, on average, $\rho \cdot C$ components, and thus the number of candidates of cardinality N_f exercised by the test are $\binom{\rho \cdot C}{N_f}$. The probability of a test failing is then

$$\Pr(t_f) = \frac{\binom{\rho \cdot C}{N_f}}{\binom{C}{N_f}} \tag{2.2}$$

A binomial coefficient can be expressed using Pochhammer's falling factorial[4]

$$\Pr(t_f) = \frac{\frac{(\rho \cdot C)_{N_f}}{N_f!}}{\frac{(C)_{N_f}}{N_f!}} = \frac{(\rho \cdot C)_{N_f}}{(C)_{N_f}} \tag{2.3}$$

As the falling factorial $(x)_n$ is equal to $\prod_{i=1}^{n}(x-i+1)$, Equation (2.3) can be rewritten as

$$\Pr(t_f) = \prod_{i=1}^{N_f} \frac{\rho \cdot C - i + 1}{C - i + 1} \tag{2.4}$$

And since $C \gg N_f$, we can approximate the value of $\Pr(t_f)$

$$\Pr(t_f) \approx \lim_{C \to +\infty} \prod_{i=1}^{N_f} \frac{\rho \cdot C - i + 1}{C - i + 1} = \rho^{N_f} \tag{2.5}$$

Then, the information gain from any given test case can be computed as demonstrated in Equation (1.9) from Section 1.3.1

$$\begin{aligned} IG &= -\Pr(t_f) \cdot \log_2(\Pr(t_f)) - \Pr(t_p) \cdot \log_2(\Pr(t_p)) \\ &= -\rho^{N_f} \cdot \log_2(\rho^{N_f}) - (1 - \rho^{N_f}) \cdot \log_2((1 - \rho^{N_f})) \end{aligned} \tag{2.6}$$

[4] http://mathworld.wolfram.com/FallingFactorial.html

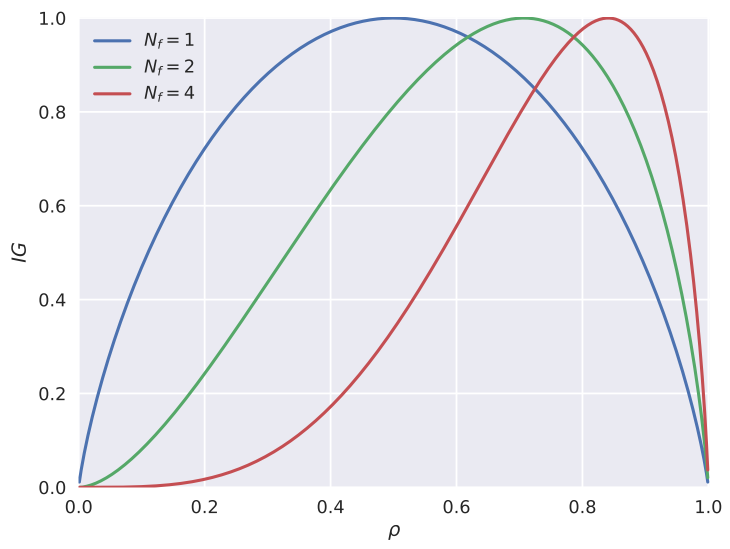


Figure 2.4: ρ versus IG for different fault cardinalities.

The optimal $IG = 1$ value corresponds to $\rho^{N_f} = 0.5$, which means that the optimal density is

$$\rho = \frac{1}{2^{\frac{1}{N_f}}} \tag{2.7}$$

Figure 2.4 shows the evolution of IG's value over the density for faults of cardinality 1, 2, and 4, where we can see a skew favoring higher densities the more components are involved in a fault. The reason for this behavior is that it is unnecessary to run sparse tests which execute less components than the number of components needed to trigger a failure. In the general case, since one does not know *a priori* about the cardinality of a failure, targeting a $\rho = 0.5$ is still the safest course of action in terms of covering all possible fault cardinalities. However, if one has a means of deducing the fault cardinality (for instance, using the defect prediction methodology as outlined in [Abreu et al., 2011]), then such information can be exploited—*e.g.*, by turning off sparse tests guaranteed to not trigger the complex fault and thus reducing the time to run the test suite.

Since $\rho = 0.5$ is our optimal (general purpose) target value, we propose a normalized metric ρ' where its upper bound (1.0) is the actual target

$$\rho' = 1 - |1 - 2 \cdot \rho| \tag{2.8}$$

and the lower bound 0 means that every cell in the matrix contains the same value. However, this optimal target is only valid assuming that all transactions in the activity matrix are *distinct*. Such assumption is not encoded in the metric itself (see Equation (1.8)). This means that a matrix with no diversity (depicted in the example from Figure 2.5a) is able to reach the ideal value for the ρ' metric.

2.3.2 Diversity

The first enhancement we propose to the ρ' analysis is to encode a check for test diversity. In a diagnostic sense, the advantage of having considerable variety in the recorded transactions is related to the fact that each diagnostic candidate's posterior probabilities of being faulty are updated with each observed transaction. If a given transaction is failing, it means that the diagnostic candidates whose components are active in that transaction are further indicted as being faulty—so their fault probability will increase. Conversely, if the transaction is passing, then it means that the candidates that are active in the transaction will be further exonerated from being faulty—and their fault probability will decrease. Having such diversity means that more diagnostic candidates will have their fault probabilities updated so that they are consistent with the observations, leading to a more accurate representation of the state of the system.

	t_1	t_2	t_3	t_4
c_1	1	1	1	1
c_2	1	1	1	1
c_3	0	0	0	0
c_4	0	0	0	0

(a) No Test Diversity.
$\rho' = 1.0 \quad \mathcal{G} = 0.0$

	t_1	t_2	t_3	t_4
c_1	1	0	1	0
c_2	1	0	1	0
c_3	0	1	1	0
c_4	0	1	0	1

(b) Test Diversity.
$\rho' = 1.0 \quad \mathcal{G} = 1.0$

Figure 2.5: Impact of diversity on ρ' and $\mathcal{G}$.

We use the Gini-Simpson index to measure diversity ($\mathcal{G}$) [Jost, 2006]. The $\mathcal{G}$ metric computes the probability of two elements selected at random being of different kinds:

$$\mathcal{G} = 1 - \frac{\sum n \times (n-1)}{N \times (N-1)} \tag{2.9}$$

where n is the number of tests that share the same activity. When $\mathcal{G} = 1$, every test has a different activity pattern. When $\mathcal{G} = 0$, all tests have equal activity. Figures 2.5a and 2.5b depict examples of repeated and diverse test cases, respectively. We can see that the ρ' metric by itself cannot distinguish between the two matrices, as they have the same density. If we also account for diversity, the two matrices can be distinguished.

2.3.3 Uniqueness

The second extension we propose has to do with checking for ambiguity in component activity patterns. If two or more components are ambiguous, like components c_1 and c_2 from the example in Figure 2.6a, then they form an *ambiguity group* (see Section 1.3.1), and it is impossible to distinguish between these components to provide a minimal diagnosis if tests t_1 and t_3 fail. As finding potential diagnostic candidates can be reduced to an *MHS* problem, breaking ambiguity groups means that more components will become inconsistent with failure observations and thus would be removed from the set of possible diagnostic candidates, improving the tractability of the Bayesian update step of the reasoning-based *SFL* approach.

	t_1	t_2	t_3	t_4
c_1	1	0	1	0
c_2	1	0	1	0
c_3	0	1	1	0
c_4	0	1	0	1

(a) Component Ambiguity.
$\rho' = 1.0 \quad \mathcal{G} = 1.0$
$\mathcal{U} = 0.75$

	t_1	t_2	t_3	t_4
c_1	1	0	1	0
c_2	1	1	0	0
c_3	0	1	1	0
c_4	0	0	1	1

(b) No Component Ambiguity.
$\rho' = 1.0 \quad \mathcal{G} = 1.0$
$\mathcal{U} = 1.0$

Figure 2.6: Impact of component ambiguity on ρ', $\mathcal{G}$ and $\mathcal{U}$.

We use a check for uniqueness ($\mathcal{U}$) as described in Equation (1.10) to quantify ambiguity. Uniqueness is also used by Baudry et al. [2006] to measure diagnosability. However, we argue that uniqueness alone does not provide sufficient insight into the suite's diagnostic ability. Particularly, it does not guarantee that component activations are combined in different ways to further exonerate or indict multiple-fault candidates. In that aspect, information regarding the diversity of a suite provides further insight.

2.3.4 Combining Diagnostic Predictors

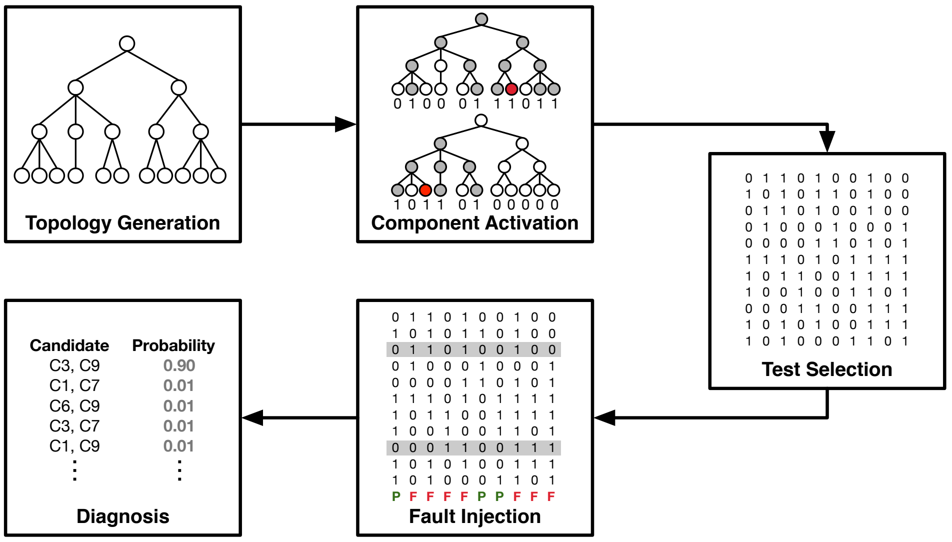
Our last step is to provide a relaxed version of entropy (which we call *DDU*) by combining the three aforementioned metrics that assess the key properties (i.e., necessary and sufficient) a coverage matrix ought to have to ensure proper diagnosability:

$$\text{DDU} = \rho' \times \mathcal{G} \times \mathcal{U} \tag{2.10}$$

and its ideal value is 1.0. We reduce ρ', $\mathcal{G}$ and $\mathcal{U}$ into a single value by means of multiplication. The reason being that since in each term the value of 0.0 corresponds to the worst-case and 1.0 to the ideal case, we are able to leverage properties of multiplication such as multiplicative identity and the zero property.

2.4 Theoretical Evaluation

A simulation approach to spectra generation enables us to consider an otherwise infeasible breadth of scenarios, so that the metric's diagnosability performance can be analyzed from a holistic, theoretical standpoint—akin to related work on spectrum based fault localization [González-Sanchez et al., 2011b; Abreu et al., 2009c; Cardoso and Abreu, 2013a]. Therefore, we first evaluate the *DDU* metric by



Candidate	Probability
C3, C9	0.90
C1, C7	0.01
C6, C9	0.01
C3, C7	0.01
C1, C9	0.01
⋮	⋮

Figure 2.7: Process followed by the spectra simulator.

generating a multitude of program spectra via simulation. This section (1) describes the topology-based spectra simulator and fault injector we created for this theoretical analysis; (2) details the experimental setup, where thousands of distinct spectra were automatically generated by the simulator; and (3) presents an assessment on the correlation of *DDU*—as well as coverage—with diagnostic effort, based on the simulated data. Afterwards, in Section 2.5, we empirically evaluate the *DDU* metric.

2.4.1 Spectra Simulator

The spectra simulator we built for this theoretical assessment is able to generate a breadth of qualitatively distinct coverage matrices. It uses *topology*-based[5] policies to select which components are active on each test, and relies on component goodnesses—as described in Section 1.2.2—to inject test failures. Figure 2.7 depicts the overall process followed by the simulator to generate a set of faulty program spectra and their respective diagnoses. The following subsections detail each step of the simulation process.

Topology Generation The first step in the simulation process is to generate a random tree with as many leaves as components to be simulated. Tree generation follows a uniform *birth-death* process, commonly used to simulate *phylogenetic trees* [Hartmann et al., 2010], in which *lineages* (or tree paths) have a constant probability of *speciating* (splitting into multiple branches), and a constant probability of going *extinct*, per time unit. The generated tree acts as the system *topology*, and is predicated on the fact that, in most programming paradigms, source code

[5]The use of topologies to generate spectra is inspired by SERG-Delft's simulator: `https://github.com/SERG-Delft/sfl-simulator`

is structured in a hierarchical fashion—especially in the case of object-oriented languages.

Component Activation After generating a topology, the component activation step generates a vast amount of test cases by activating components and propagating these activations through the topology. This step starts with the selection of a component (which we call the *anchor*) and setting it as active in a newly created test-case. Anchor components are shown as red tree nodes in the Component Activation step depicted in Figure 2.7. With the selection of an anchor, we randomly activate other components conditioned upon their distance to the anchor—following the assumption that the farther away two components are, the less related they are and hence less likely to be covered in the current test-case being generated. Worth noting that coverage density of a test can be manipulated by multiplying the activation probability by a *density term*. If this term is < 1, then sparse test cases are generated. Conversely, a value > 1 yields denser test cases. We generate test cases using a wide spectrum of *density terms*.

The component activation process is repeated numerous times for each component in the system, so that a large collection of test cases is available to the next steps in the simulation.

Test Selection This step consists on selecting a set of test cases out of the test case pool generated in the previous step. We have chosen to select as many tests as there are components in the system—yielding square coverage matrices. Having the test suite depend on the number of components allows it to (reasonably) grow with program size, with the assumption that the larger the code base is, the more tests are created.

Fault Injection For each matrix that the previous step produces, we inject it with: (1) a single fault, (2) multiple independent faults, and (3) multiple dependent faults. In the first case, we randomly assign a component from the system as the faulty one, and set each test which covers the faulty component as having a failing outcome. In scenario (2), multiple components are considered as being faulty, and thus tests that cover any non-empty subset of faulty components are set to failing. In the last scenario, only tests that cover the conjunction of all failing components are set to failing. We include multiple-faulted scenarios in our analysis since, as will be discussed in Chapter 3, such scenarios account for a non-trivial portion (20%) of bug-fixing tasks in open-source projects.

The fault injection step is also able to consider component goodnesses, which, as detailed in Section 1.2.2, describe the probability of a faulty component exhibiting nominal behavior (and thus not triggering a test failure).

Diagnosis We diagnose the faulty spectra generated in the previous step using the reasoning-based *SFL* technique described in Section 1.2.2.

2.4.2 Setup

We have run our simulation 20 times so that 20 distinct topologies ranging from 100 to 500 components were considered. For each topology, all components acted as anchors, generating a test-case pool using several *density terms*. Each test-case pool produced 100 matrices, which were fault-injected—with a single fault, two/three independent faults, and two/three dependent faults. Goodnesses ranged from 0 to 0.25. Regarding metrics, we have gathered coverage, *DDU*, entropy, and effort to diagnose for every faulty spectra generated by the simulator.

To ensure reproducibility, our spectra simulator, and its respective configuration file describing this experiment, are made available[6]. In total, more than 300,000 spectra were simulated, fault injected, and diagnosed in this experiment.

2.4.3 Results

Below we present and discuss the spectra simulation results by (1) evaluating the diagnostic quality using the effort metric as described in Section 1.2.3, and by (2) evaluating the simulated spectra's propensity for error detection.

Diagnostic Quality Diagnostic effort results for every spectrum generated in this experiment are shown in Figures 2.8 to 2.11. Each figure shows a scatter plot portraying the relation of diagnostic effort[7] with different metrics—namely coverage, *DDU*, entropy, and the average of density, diversity and uniqueness. Points in the scatter plot represent simulated spectra. Beside each scatter plot are three two-dimensional histograms depicting the distribution of spectra containing each fault type.

Figure 2.8 portrays the relation between coverage and diagnostic effort for all simulated spectra. Regarding spectra that were injected with a single fault, their diagnosability improves by increasing coverage. Note that single-faulted spectra seem to form several downward lines in the scatter plot—each of these lines corresponds to a different topology used as the basis for emulating software structure. We can therefore make two observations. The first is that, for a given topology, the selection and composition of the test suite influences not only coverage but also the effort to diagnose. The second is that the choice of base topology also influences diagnostic quality.

[6]Available at `https://github.com/aperez/sfl-simulator`.

[7]Normalized over the number of components, so that spectra of systems with a different number of components can effectively be compared.

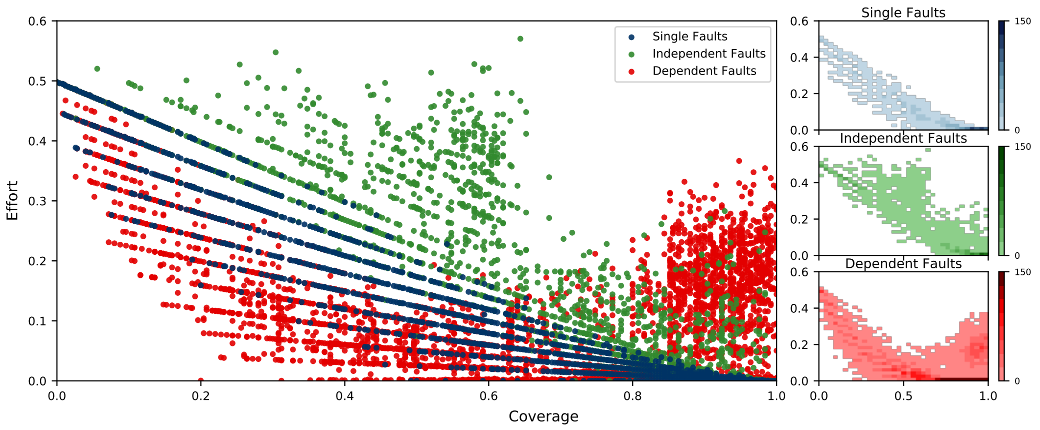


Figure 2.8: Relation between diagnostic effort and coverage.

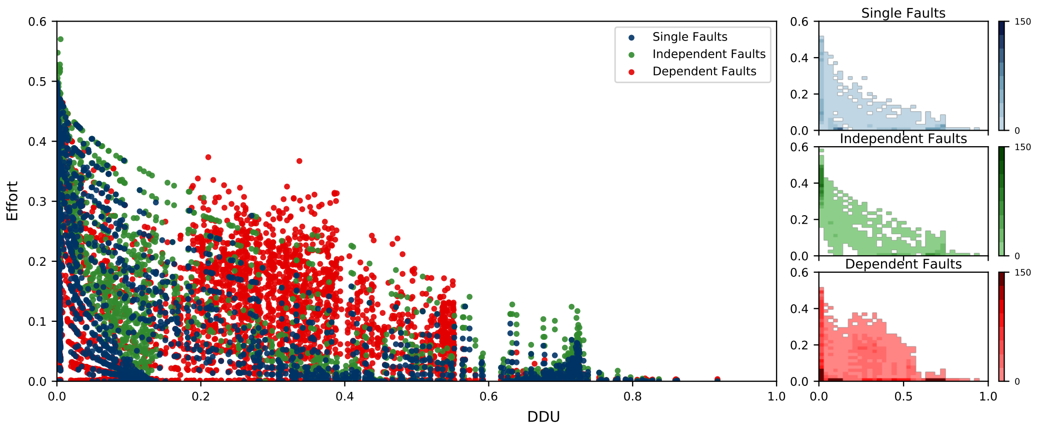


Figure 2.9: Relation between diagnostic effort and *DDU*.

While single-fault diagnostic effort decreases with coverage, the same cannot be said for scenarios with multiple faults, especially ones where dependent faults were injected, since several spectrum instances with high coverage are not in the bottom-right of the plot. For these scenarios, high coverage is not a good indicator of diagnosability. An illustrative example of such phenomenon is as follows. Consider a spectrum that resembles a diagonal matrix, where each test exercises a single distinct component. Such a spectrum has high coverage—because every component is exercised—and, at the same, is *sparse*—since all tests contain a single component activation. In effect, this is analogous to a high-coverage *unit* test suite with no integration tests exercising multiple components. For single fault scenarios, this suite is very likely to find and accurately isolate faults. However, in cases where a fault requires set of component activations for an error to be triggered, this suite cannot provide enough evidence for fault localization algorithms to pinpoint faults.

Figure 2.9 depicts the relation between *DDU* and effort. We can tell that this metric upper bounds the effort to diagnose —the higher the *DDU*, the lower the maximal diagnostic effort —providing a more accurate expectation of diagnosability when

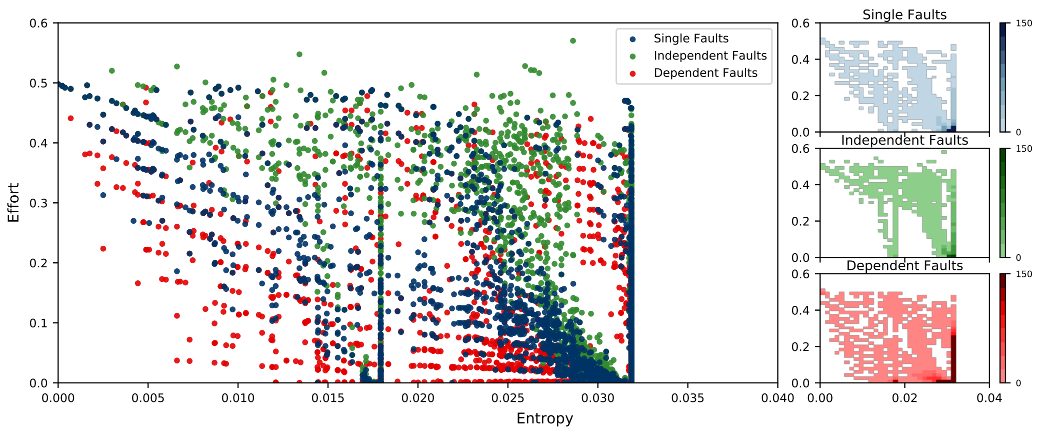


Figure 2.10: Relation between diagnostic effort and entropy.

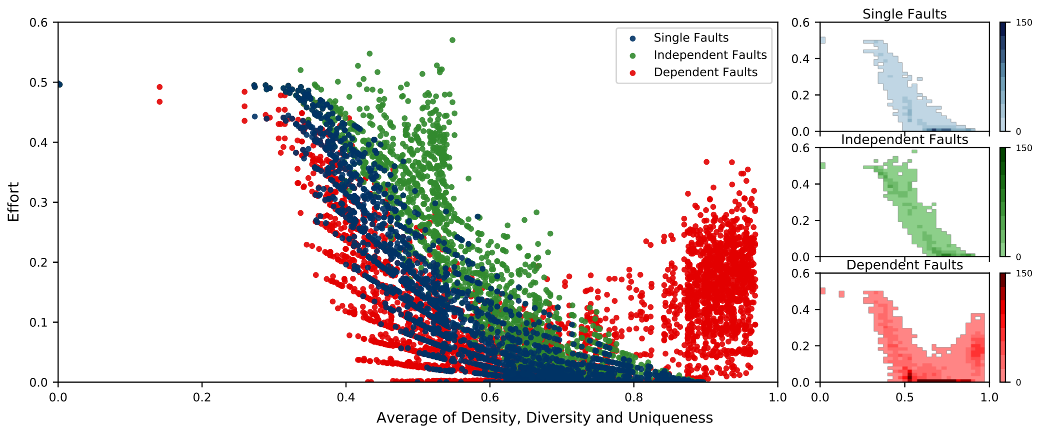
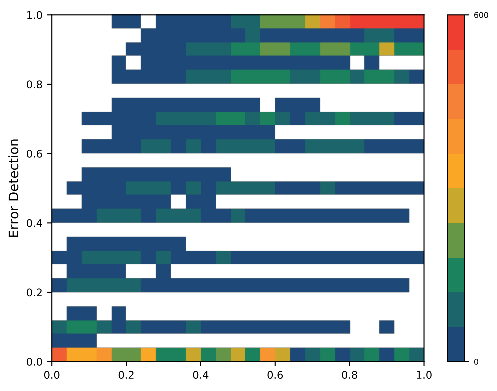


Figure 2.11: Relation between diagnostic effort and the average of density, diversity and uniqueness.

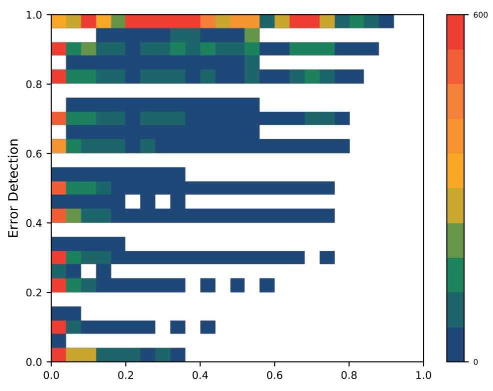
compared with coverage. As opposed to coverage, multiple faults do not negatively influence the *DDU*'s diagnostic accuracy.

Figure 2.10 shows test entropy—as described in Section 2.3—in the x-axis. Note that entropy values range from 0 to 1, but to improve legibility, we are showing a partial range of entropy values up to 0.04 as no spectra in our simulation exceeded this value. In effect, the number of test cases generated by the simulator (set to be the same as the number of components in every generated spectra) is insufficient to significantly explore the entire range of entropy values. Limiting the number of tests was an intentional way to model how developers test in practice, and therefore it leads us to conclude that optimizing for entropy is infeasible with a reasonable number of tests.

We discuss in Section 2.3.4 the reasons for choosing multiplication as a way of reducing the composing terms of *DDU* (namely density, diversity and uniqueness) into a single value that represents the system's diagnosability. While we explain why each term is important for the overall diagnosability, it might be the case that there



(a) Coverage.



(b) *DDU*.

Figure 2.12: Two-dimensional histograms depicting the number of simulated matrices along with the relation between error detection and several metrics.

is a better way to reduce them into a one-dimensional value. Figure 2.11 depicts using the average of density, diversity and uniqueness values as the measure for diagnosability—as opposed to their multiplication, which is depicted in Figure 2.9. We can conclude that the multiplication of density, diversity and uniqueness more accurately predicts the diagnostic performance of the test suite.

Error Detection Besides investigating diagnostic quality, which relates to the actual effort bugs take to be located, we have also recorded the *error detection rate*. This evaluates the propensity for faults in a given coverage matrix to induce test errors, and is achieved by keeping track of the frequency in which errors are detected in faulty spectra. Each coverage matrix we generate is subject to multiple rounds of fault injection. As a result, we generate sets of spectra that exhibit the same coverage matrix and different error vectors. Error detection rate is then the frequency by which these sets of spectra exhibit failing error vectors.

Figures 2.12a and 2.12b show two-dimensional histograms depicting the error detection frequency of coverage matrices along coverage values and *DDU* values, respectively. Figure 2.12a tells us that the majority of high coverage spectra are able to produce test failures when faults are injected, as portrayed by the intensity of the top-right portion of the histogram. However, we still observe a significant portion of cases with low error detection despite their coverage value, as evidenced by the intensity of the bottom row in the histogram. Such spectra do not have adequate test cases that *detect* the injected faults. In contrast, we see that when *DDU* is considered—Figure 2.12b —, there are considerably less cases of high-*DDU* spectra yielding low error detection rates. This is initial evidence that *DDU* may be suited for measuring the *adequacy* of test suites, besides simply measuring diagnosability.

2.5 Empirical Evaluation

Results obtained by simulating a breadth of program spectra seem to indicate that, from a theoretical standpoint, *DDU* effectively estimates the diagnostic effort required to pinpoint bugs, regardless of fault type. However, these promising results do not exclude the need to evaluate the metric against real-world subjects. This section details our following experiment, in which we empirically evaluate the proposed metric in regard to its ability to assess diagnostic quality. We aim to address the following research questions:

Research Question 2.1

Is the *DDU* metric more accurate than the state-of-the-art in diagnosability assessment?

Research Question 2.2

How close does the *DDU* metric come to the (ideal yet intractable) full entropy?

Research Question 2.3

Does optimizing a test-suite with regard to *DDU* result in better diagnosability than optimizing adequacy metrics such as branch-coverage in traditional scenarios?

RQ2.1 asks if there is a benefit in utilizing the proposed approach as opposed to density and uniqueness—which have been used in related work. **RQ2.2** is concerned with assessing if *DDU* shares a statistical relationship with entropy—the measurement whose maximal value describes an optimal (yet intractable and impractical) coverage matrix. **RQ2.3** asks if using *DDU* as an indicator of the diagnostic ability of a test-suite is more accurate than using standard adequacy measurements like branch-coverage in a setting with real faults.

2.5.1 Experimental Setup

Our empirical evaluation compares *DDU* to several metrics in use today. To effectively compare the diagnosability of test-suites of a given program that maximize a specific metric, we leverage a test-generation approach. EVOSUITE[8] is a tool that employs search-based testing approaches to create new test cases [Fraser and Arcuri, 2011]. It applies a **Genetic Algorithm (*GA*)** to minimize a *fitness function* which describes the distance to an optimal solution. The metrics to be compared are *DDU*—our proposed measurement; density and uniqueness to be able to answer **RQ2.1**; entropy to

[8]EVOSUITE tool is available at `http://www.evosuite.org`. Version 1.0.2 was used for experiments.

answer **RQ2.2** and lastly branch-coverage for **RQ2.3**. These metrics were encoded as *fitness functions* in the EVOSUITE framework. As the *GA* in EVOSUITE tries to minimize the value of a function over a test suite TS, the *fitness functions* for each metric $\mathcal{M}$ are as follows

$$f_{\mathcal{M}}(TS) = |\mathcal{O}_{\mathcal{M}} - \mathcal{M}(TS)| \tag{2.11}$$

where $\mathcal{O}_{\mathcal{M}}$ is the optimal value of metric M (*e.g.,* 1.0 for the case of branch-coverage, and 0.5 for density), and $M(TS)$ is the result of applying metric M to test suite TS. To account for the randomness of EVOSUITE's *GA*, we repeated each test-suite generation experiment 10 times. EVOSUITE's maximum search time budget was set to 600 seconds, which follows the setup of previous studies also using the tool (*e.g.*, [Campos et al., 2013]).

EVOSUITE by itself does not generate fault-finding *oracles*—otherwise, a model of correct behavior would have to be provided. Instead, it creates assertions based on static and dynamic analyses of the project's source code. This means that if we run the generated test-suite against the same source code used for said generation, all tests will pass (provided the code is deterministic[9]). Thus, if the source code submitted for test-generation contains faults, no generated test oracle will expose them.

For the experiments comparing with the state-of-the-art and the idealistic approach (to answer **RQ2.1** and **RQ2.2**, respectively), we need a controlled environment so that oracle quality (which in itself is an orthogonal factor) does not affect results. Therefore, the experiment described in Section 2.5.2 mutates the program spectrum of generated test-suites to contain seeded faults and seeded failing tests, akin to the fault injection step of our spectra simulator (*cf*. Section 2.4.1). In each experiment, a set of components were considered as faulty, and tests that exercise them were set as failing according to a *goodness probability*—we have set faulty component *goodnesses* to 0.25, meaning that whenever a faulty component is involved in a test, there is a 75% chance that the test will be set as failing. The chosen value is a compromise between perfect error detection (*i.e.*, *goodness* of 0) and essentially random error detection (*goodness* of 0.5). This fault injection approach is common practice among controlled, theoretical evaluations of spectrum-based diagnoses [González-Sanchez et al., 2011b; Abreu et al., 2009b].

For assessing the applicability in real world scenarios, and to answer **RQ2.3**, we need real life bugs and fixes. Therefore, in Section 2.5.3 we make use of DEFECTS4J[10]—a software fault catalog [Just et al., 2014a]—to generate test-suites from fixed versions

[9] EVOSUITE also tries to replicate the state of the environment at each test-run so that even some non-deterministic functionality such as random number generation can be tested.

[10] DEFECTS4J tool is available at `https://github.com/rjust/defects4j`. Version 1.0.1 was used for experiments.

of a program and then gather program spectra by testing the corresponding faulty version.

Spectrum gathering was performed at the branch granularity for both experiments, so every component in our subjects' coverage matrices corresponds to a method branch—this way we can fairly compare our approach to branch coverage. Each program spectrum gathered in the previous step is then diagnosed using the automated diagnosis tool CROWBAR[11]. This tool implements the spectrum-based reasoning approach, and generates a ranked list of diagnostic candidates for the observed failures.

For a given subject program, to compare the diagnosability of a test-suite generated by the *DDU* criterion with the one generated by a criterion C, we use the following metric

$$\Delta_{\text{Effort}}(C) = \text{Effort}_C - \text{Effort}_{\text{DDU}} \tag{2.12}$$

where $\text{Effort}_{\text{DDU}}$ is the effort to diagnose using the test-suite generated with the *DDU* criterion and Effort_C is the effort to diagnose with the test suite by maximizing some criterion C. Effort takes as input the ranked list of diagnostic candidates from CROWBAR and estimates quality of diagnosis as described in Section 1.2.3. The $\Delta_{\text{Effort}}(C)$ metric ranges from -1 to 1. Positive values of $\Delta_{\text{Effort}}(C)$ mean that the bug is found faster in diagnoses that use the *DDU* generated test suite. Negative values mean that the faulty component is ranked higher in the C-generated test-suite than the *DDU* one, thus requiring less spurious diagnostic inspections. $\Delta_{\text{Effort}}(C)$ of value 0 means that the faulty component is ranked with the same priority in both test generations.

We make use of kernel density estimation plots to show the $\Delta_{\text{Effort}}(C)$ values in Figures 2.13 and 2.14. Such plots estimate the probability density function of a variable, *i.e.*, they describe the relative likelihood (y-axis) for a random variable ($\Delta_{\text{Effort}}(C)$ in our case) to take on a given value (x-axis). In our experiments, the higher the density value at a certain value in the x-axis, the more instances with $\Delta_{\text{Effort}}(C)$ near that value were observed. Note that the observed data is shown as a *rug plot*, with tick marks along the x-axis (reminiscent of the tassels on a rug).

2.5.2 Diagnosing Seeded Faults

Our first experiment attempts to answer **RQ2.1** and **RQ2.2** by generating test-suites and seeding faults in their spectra in a controlled way. We same set of subjects as empirical evaluations from related work [Campos et al., 2013]. Namely, we use the open-source projects Apache Commons-Codec, Apache Commons-Compress, Apache Commons-Math and JodaTime. For each subject, we generate test-suites that

[11]CROWBAR tool is available at `https://github.com/TQRG/crowbar-maven-plugin`.

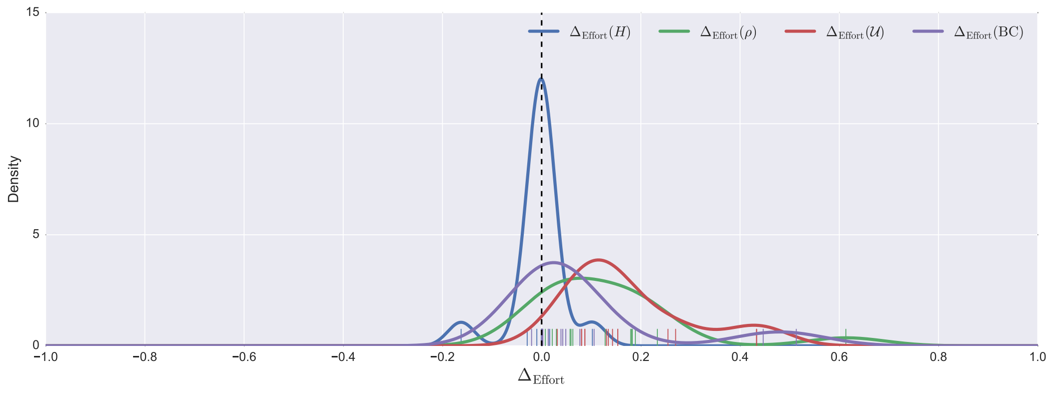


Figure 2.13: Kernel density estimation of seeded fault experiment. Entropy generation criterion shows similar diagnostic accuracy when compared *DDU*. The remaining generation criteria exhibit worse diagnostic performance than *DDU*.

optimize *DDU*, branch-coverage, entropy, density, and uniqueness. In total, 1050 program spectra were generated and diagnosed.

Experimental results are shown in Figure 2.13. When we consider the entropy generation, we can say that the resulting test-suites are very similar in terms of diagnosability compared to *DDU*, since $\Delta_{\text{Effort}}(H)$ is denser at the origin. For the remaining generation criteria, their respective Δ_{Effort} probability masses are shifted to $\Delta_{\text{Effort}} > 0$, so their diagnostic reports perform worse at diagnosing the faults than when *DDU* is utilized. In fact, our inspection of experimental results reveals that, when optimizing branch-coverage, 78% of scenarios showed lower diagnostic accuracy when compared to *DDU*. For both the density-optimized and uniqueness-optimized test generations—which are the state-of-the-art measurements for test-suite diagnosability—this figure rises to 100% of scenarios.

We show in Table 2.1 the dominant metric median values for each generation criterion along with the median number of tests generated. By dominant metric we mean the metric which that particular test generation was trying to optimize. Along with the median value we also show (where available) the metric's Pearson correlation with entropy (denoted by r_H) and the p-value of the correlation. With 95% confidence, we can say that the correlation values shown are statistically significant. *DDU* exhibits a high correlation with entropy, having $r_H > 0.95$ for all subjects. In all other generation criteria, the correlation with entropy fluctuates considerably between subjects. Also, note that for both ρ and branch-coverage criteria, their dominant mean values approach the theoretical optima (at 0.5 and 1.0, respectively) while Δ_{Effort} still shows that *DDU* test generation was able to produce suites with better diagnostic accuracy.

Revisiting the first research question:

Table 2.1: Metric results for the seeded faults experiment.

Subject	Median / Size / Correlation / Correlation **p-value**				
	H	**DDU**	ρ	$\mathcal{U}$	**BC**
Apache Commons-Codec	2.65×10^{-2}	0.620	0.476	0.669	0.910
	177	170	126	81	177
	N.A.	0.957	0.658	0.902	0.793
	N.A.	2.71×10^{-3}	1.98×10^{-2}	3.58×10^{-2}	2.08×10^{-3}
Apache Commons-Compress	4.66×10^{-2}	0.962	0.510	0.669	0.825
	108	108	30.5	29.5	126
	N.A.	0.999	0.999	0.873	0.968
	N.A.	1.08×10^{-6}	7.51×10^{-7}	1.47×10^{-3}	9.62×10^{-4}
Apache Commons-Math	4.36×10^{-2}	0.818	0.424	0.659	0.922
	497	467	402	246	528.5
	N.A.	0.989	0.905	0.725	0.885
	N.A.	4.68×10^{-4}	1.85×10^{-2}	4.79×10^{-2}	2.31×10^{-2}
JodaTime	1.580×10^{-2}	0.582	0.369	0.417	0.790
	265	265	267	171	267
	N.A.	0.976	0.674	0.921	0.654
	N.A.	8.54×10^{-4}	1.60×10^{-2}	2.59×10^{-2}	2.09×10^{-2}

Research Question 2.1

Is the *DDU* metric more accurate than the state-of-the-art in diagnosability assessment?

Answer: There is a clear benefit in optimizing a suite with regard to *DDU* compared to density if we consider the effort of finding faults in a system. This is evidenced by the fact that 100% of scenarios in our seeded fault experiment show improved diagnostic accuracy when using *DDU* when compared to the state-of-the-art density and uniqueness measurements.

If we look at the second research question:

Research Question 2.2

How close does the *DDU* metric come to the (ideal yet intractable) full entropy?

Answer: Table 2.1 shows a strong correlation between entropy and *DDU*, with a Pearson correlation value above 0.95 for all subjects. Correlation of other metrics is much lower and varies greatly across subjects. Thus, we can conclude that *DDU* closely captures the characteristics of entropy.

The reader might then pose the question: if maximal entropy does indeed correspond to the optimal coverage matrix, why should one avoid using it as the diagnosability metric? While we agree that in automated test generation settings entropy can be

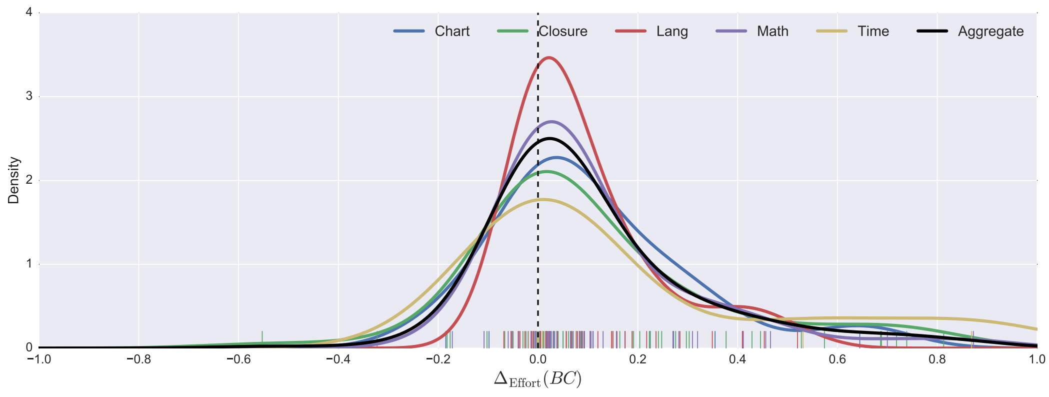


Figure 2.14: Kernel density estimation of the $\Delta_{\text{Effort}}(\text{BC})$ metric for DEFECTS4J subjects. 77% of instances have a positive $\Delta_{\text{Effort}}(\text{BC})$, meaning that branch-coverage generations perform worse than *DDU* generations.

plugged as the fitness function to optimize[12], for manual test generation entropy will yield very small values for any complex system, as one can see from Table 2.1. In fact, for a system composed of only 30 components, the number of tests needed to reach entropy of 1.0 surpasses the billion mark. This makes it difficult for developers to leverage information out of their test-suite's entropy value to gauge when can one confidently stop writing further tests.

2.5.3 Diagnosing Real Faults

We used the DEFECTS4J database [Just et al., 2014a] for sourcing the experimental subjects. DEFECTS4J is a database and framework that contains real software bugs from open source projects. For each bug, the framework provides faulty and fixed versions of the program, a test suite exposing the bug, and the fault location in the code. The idea behind DEFECTS4J is to allow for reproducible research in software testing using real-world examples of bugs, rather than using the more common hand-seeded faults or mutants. In our evaluation, we generate test suites for each of DEFECTS4J's 357 catalogued bugs, using both branch-coverage and *DDU* as EVOSUITE's fitness functions, and then compare the two generated suites with regard to their diagnosability and adequacy. The experiments' methodology is as follows. For every bug in DEFECTS4J's catalog, we use EVOSUITE to generate test suites for the fixed version of the program. The test suites are executed against the faulty program versions. This means that any test failure is due to the bug—which is the delta between the faulty and fixed program versions.

Out of the 357 catalogued bugs in DEFECTS4J (version 1.0.1), not all were considered for analysis. Scenarios were discarded due to the following reasons:

- EVOSUITE returned an empty suite;

[12]Because tools like EVOSUITE can be configured with a time budget as another stopping criteria.

Table 2.2: DEFECTS4J Projects.

Identifier	Project Name	# Scenarios	Considered
Chart	JFreechart	26	1, 4, 6, 8–11, 13–15, 18, 20, 22, 24, 26
Closure	Closure Compiler	133	3, 4, 7, 9, 12, 14–17, 19, 20–28, 30, 33–35, 39, 43, 44, 46–49, 51, 52, 54–56, 58, 63, 65, 66, 67, 69, 71–74, 76–78, 82, 85, 87, 107, 108, 110–113, 115, 116, 118, 119, 124, 126, 127, 129–132
Lang	Apache Commons-Lang	65	1–7, 9–14, 16, 17, 19, 21, 22, 24–28, 30, 31, 33, 36, 38–42, 46, 47, 49, 50–57, 59–61, 65
Math	Apache Commons-Math	106	1–10, 14–16, 18–20, 24–27, 29, 30, 32, 34, 35, 37–42, 44–46, 48–56, 100, 101, 103, 105, 106
Time	JodaTime	27	6, 8, 12, 15, 21, 22, 26, 27

Table 2.3: Metric medians and statistical tests.

	Branch-Coverage Generation	**DDU Generation**
Branch Coverage	0.85	0.75
DDU	0.10	0.42
Suite Size	291	374
Effort	0.31	0.10
Shapiro-Wilk	$W = 0.92$ p-value $= 1.70{\times}10^{-8}$	$W = 0.85$ p-value $= 1.05{\times}10^{-12}$
Wilcoxon Signed-rank	$Z = 2335.0$ p-value $= 3.50{\times}10^{-13}$	

- The generated suite did not compile or produced a runtime error;
- No failing tests were present in either *DDU* or branch-coverage criteria for generating test suites.

In total, 171 scenarios were filtered out. The remaining 186 listed in Table 2.2 are fit for analysis and their results are used throughout this section.

Experimental results are shown in Figure 2.14. Results are shown per-subject. We can see that for every subject in the DEFECTS4J catalog, all their estimated probability density funtions are shifted towards $\Delta_{\text{Effort}}(\mathrm{BC}) > 0$, meaning that the majority of instances have better diagnostic accuracy when test generation optimizes *DDU*. In fact, our experiments reveal that 77% of scenarios (144 in total) yield a positive $\Delta_{\text{Effort}}(\mathrm{BC})$.

We performed statistical tests to assess whether the gathered metrics yielded statistically significant results. Table 2.3 shows the relevant statistics. The first four rows show the median values for branch-coverage, *DDU*, generated suite size and diagnosis effort for both EVOSUITE test generations. As to be expected, the median branch-coverage is higher in the branch-coverage-maximizing generation. Conversely, the *DDU* criterion yields the higher *DDU*. Results in the effort row corroborate our observations from Figure 2.14—the test suites optimizing *DDU* take on average less effort to diagnose the fault. In fact, our results show that the effort reduction when considering *DDU* over branch-coverage is 34% on average. However, this fact alone does not guarantee that the results are significant, which prompted us to perform statistical tests. The first test performed was the Shapiro-Wilk test for normality [Shapiro and Wilk, 1965] of effort data for both generations. The results, which can be seen in the fourth row of Table 2.3, tell us that the distributions are not normal, with confidence of 99%.

Given that the effort data is not normally distributed and that each observation is paired, we use the non-parametrical statistical hypothesis test Wilcoxon signed-rank [Wilcoxon, 1945]. Our null-hypothesis is that the median difference between the two observations (*i.e.*, Δ_{Effort}) is zero. The fifth row in Table 2.3 shows the resulting Z statistic and p-value. With 99% confidence, we can refute the null-hypothesis. Revisiting **RQ2.3**:

Research Question 2.3

Does optimizing a test-suite with regard to *DDU* result in better diagnosability than optimizing adequacy metrics such as branch-coverage in traditional scenarios?

Answer: Since the median effort in the *DDU* generation is lower—the reduction amounting to 34% on average—we can say that optimizing for *DDU* produces better, statistically significant, diagnoses when compared to test suites that optimize for branch-coverage.

2.5.4 Threats to Validity

We now outline the potential threats to validity of the experiment detailed above:

External Validity When choosing the projects for our study, our aim was to opt for projects that resemble a general large-sized application being worked on by several people. To reduce selection bias and facilitate the comparison of our results, we decided to use the same experimental subjects from related work (such as [Campos et al., 2013]), as well as the real-world scenarios compiled in the DEFECTS4J database. Another threat to external validity relates to the choice of test suites generated by EVOSUITE. Additional research is needed to see how the metric behaves both with different test-generation frameworks (such as RANDOOP [Pacheco and Ernst, 2007]) and with hand-written test cases.

Construct Validity A potential threat to construct validity relates to the choice of effort as indicator for diagnosability. However, as argued in Section 1.2.3 this choice reflects the effort that a programmer with minimal knowledge about the system would require to effectively pinpoint all the faults that explain the observed failures.

Internal Validity The main threat to internal validity lies in the complexity of several of the tools used in our experiments, most notably the *DDU* test generator and our diagnosis tool.

2.6 Discussion

DDU was shown to be useful for evaluating the quality of a test-suite. But what are the practical implications of this finding? We outline such assessments next.

Composition of a Test Suite We argue that the *DDU* analysis can suggest an ideal balance between unit tests and system tests (*i.e.,* when *DDU* reaches its optimal value) due to its density term. We are then able to compare the balanced suites to ones created following testing practices currently established at software development companies. For instance, Google suggests a 70%/20%/10% split between unit, system and end-to-end tests in a suite[13]. Is this split indeed ideal in terms of diagnostic accuracy? We believe a *DDU* analysis can provide guidance as to what the answer is, as evidenced in the theoretical evaluation. Our simulation of spectra shows that changing the composition of a test suite through test selection does impact the diagnostic effectiveness for a given base topology, and as such, an optimal selection can be achieved through maximization of the *DDU* metric.

Test Design Strategy We expect the *DDU* analysis to be used as the first step of a test design strategy that aims to increase diagnostic accuracy of a suite. For that, we envision that new test patterns that focus on optimizing diagnosability will need to be researched and incorporated in established test strategy corpora such as that of Binder [2000].

Additionally, an *ensemble* of strategies that individually improve *DDU*'s density, diversity, and uniqueness terms could also be considered. Density-based test strategies would focus on selecting the optimal test scope. Diversity-based strategies would focus on identifying and exercising untested code paths. Uniqueness-based strategies would focus on decoupling component executions. Tying into genetic-algorithm-based automated test generation tools such as the one used in our evaluation, and

[13]Google Testing blog: Just Say No to More End-to-End Tests. `http://goo.gl/S5HhZ7`.

with the rising interest in *explainable artificial intelligence*[14], these three strategies could serve as the cornerstone for a multi-objective approach to test generation that would not only recommend the best test to add to an existing test suite, but also provide a *reasoned explanation* as to why that test would benefit and improve the system's ability to diagnose eventual faults.

At a broader scope, our simulation experiment also tells us that system structure, or architecture—which we call topology—also has an influence on diagnosability. Test design strategies will necessarily need to utilize such structural information to provide better assessments as to what tests should be performed to improve diagnostic quality. Conversely, it is also not unreasonable to expect that a change in the system's structure could yield considerable gains in diagnosability.

Visualization In coverage metrics, it is straightforward to visualize the analysis of a system so that users know what code components were left untested, highlighting where to focus when writing new test cases. Is there a way to visualize *DDU* analysis in a similar way? In our opinion, the challenge for creating such visualization would be conveying the three different properties that the *DDU* metric captures in such a way that would elucidate users regarding what their best next action is in order to increase the system's diagnosability. We envision that visualization approaches for program comprehension, such as EXTRAVIS [Cornelissen et al., 2011], will constitute a solid starting point for a study on visual, interactive and actionable ways to improve the system's diagnosability.

Generalization to Other Debugging Techniques We show that *DDU* depicts the ability to diagnose of *SFL* approaches. However, our intuition is that *DDU* is general and applies to any diagnosis technique that uses a failing test suite as the basis for locating faults. We plan to investigate this hypothesis as future work.

Adequacy Assessment *DDU* provides an assessment of the diagnostic effectiveness of a given test suite. It remains to be seen if that can also be said for assessing the fault finding effectiveness, which is also a good avenue for future work. In the meantime, we consider our metric to be a complement to adequacy metrics, and envision that testers will employ a hybrid approach to evaluating test quality that relies on branch coverage and *DDU* to assess adequacy and diagnosability, respectively.

[14]Such as DARPA's Explainable Artificial Intelligence (XAI) program: `https://www.darpa.mil/program/explainable-artificial-intelligence`

2.7 Related Work

Related work in the assessment of the diagnosability of a test suite has focused on three key areas: test-suite minimization and generation strategies, and assessing oracle quality.

The topic of test-suite minimization is a prime candidate for our approach, since it has been shown that there is a tradeoff between reducing tests and the suite's fault localization effectiveness [Yu et al., 2008]. In minimization settings, one tries to reduce the number of tests (and thus its overall running time) while still ensuring that an adequacy criterion—usually branch coverage—is not greatly affected. Current minimization strategies can often improve the diversity score of a coverage matrix by removing tests with identical coverage patterns [Gong et al., 2012] at the cost of overlooking density and uniqueness, which we argue are of key importance to assess diagnosability. The uniqueness property is also exploited by Xuan and Monperrus [2014], with a *test-case purification* approach that separates a test-case into multiple smaller tests. This approach overlooks the fact that density will decrease, along with the ability to diagnose a multiple-faulted scenario.

Current test-suite minimization frameworks that take adequacy criteria into account could also benefit from our approach to preserve diagnostic accuracy if a multi-objective optimization (such as, *e.g.*, [Yoo and Harman, 2010; Alipour et al., 2016]) to also account for *DDU* is employed. This paves an interesting avenue for future work.

On the test-suite generation front, previous work has also started considering diagnosability as a generation criterion. The work of Campos et al. [2013], which generated tests that would converge towards coverage matrix densities of 0.5, has paved the way for creating improved measurements like *DDU*. Checks for diversity and uniqueness were not explicitly added, and we show when we answer **RQ2.1** in Section 2.5 that the density criterion produces results that are less diagnostically accurate. Another approach to suite generation is one by Artzi et al. [2010], that proposes an online approach that leverages *concolic analysis* to generate tests that are similar to existing failing tests in a system.

Lastly, we highlight some of the work targeting diagnosability by improving test oracle accuracy. Schuler and Zeller [2011] propose *checked coverage* as a way of assessing oracle quality. *Checked coverage* tries to gauge whether the computed results from a test are actually being checked by the oracle. X. Wang et al. [2009] have proposed a way of addressing *coincidental correctness*—when a fault is executed but no failure is detected—by analyzing data and control-flow patterns. Just et al. [2014b] investigated the use of mutants to estimate oracle quality, and compared their performance against the use of real faults. Their results suggest that a suite's mutation score is a better predictor of fault detection than code coverage. We consider this topic of assessing and improving oracle quality of critical importance towards test-suite diagnosability, but also orthogonal to *DDU* in that the two would complement each other.

2.8 Summary

In this chapter, we addressed the limitations of current diagnosability measurements outlined in Section 1.3.1. Concretely, in this chapter:

- We proposed the *DDU* diagnosability measurement which assesses fault localization effort given a faulty test suite (Section 2.3, page 24). *DDU* leverages and combines three core diagnosability traits:
    1. Density of the activity matrix (Section 2.3.1, page 24);
    2. Diversity of the test suite (Section 2.3.2, page 27); and
    3. Uniqueness of component coverage patterns (Section 2.3.3, page 27).
- We conducted a theoretical evaluation using simulated (faulty) spectra to evaluate the full range of possible *DDU* values (Section 2.4, page 28). Results show that higher *DDU* values correlate to lower maximal diagnostic effort.
- We conducted an empirical evaluation on real programs, using the EVOSUITE test generation tool to generate test suites maximizing different software metrics, namely *DDU*, branch-coverage, entropy, *etc* (Section 2.5, page 35). Results show that:
    1. Test suites generated by maximizing *DDU* exhibit lower effort to diagnose compared to test suites generated by maximizing current state-of-the-art diagnosability metrics such as density or uniqueness (thus answering **RQ2.1**, page 39);
    2. *DDU* performance is correlated with the (optimal, but intractable) entropy measurement, showing that it effectively captures the fault-revealing properties of entropy (answering **RQ2.2**, page 39);
    3. Test suites that maximize *DDU* yield a 34% average reduction in diagnostic effort compared with test suites that maximize branch-coverage (for the same Evosuite generation time budget) showing that *DDU* is a more accurate diagnosability predictor than coverage (answering **RQ2.3**, page 42).

3 Prevalence of Single-Fault Fixes Throughout Development

Prevalence of Single-Fault Fixes and Its Impact on Fault Localization
Alexandre Perez, Rui Abreu, and Marcelo d'Amorim
In: Proceedings of the IEEE International Conference on Software Testing, Verification and Validation, ICST'17, Tokyo, Japan, March 13-17, pp. 12–22, 2017.

Abstract *Several fault predictors were proposed in the context of similarity-based SFL approaches to rank software components in order of suspiciousness of being the root-cause of observed failures. Previous work has also shown that some of the fault predictors (near-)optimally rank software components, provided that there is one fault in the system. Despite this, further work is being spent on creating more complex, computationally expensive, model-based techniques that can handle multiple-faulted scenarios accurately. However, our hypothesis is that when software is being developed, bugs arise one-at-a-time and therefore can be considered as single-faulted scenarios. We describe an approach to mine repositories, find bug-fixes, and catalog them according to the number of faults they fix, to assess the prevalence of single-fault fixes. Our empirical study using 279 open-source projects reveals that there is a prevalence of single-fault fixes, with over 82% of all fixes only eliminating one bug from the system, enabling the use of simpler, (near-)optimal, fault predictors.*

3.1 Introduction

When considering similarity-based *SFL* approaches (see Section 1.2.1), the O predictor, proposed by Abreu et al. [2009c] and by Naish et al. [2011], has been shown to be the *optimal* heuristic for locating faults, provided the system under analysis contained only one fault. However, in the eventuality of the system containing multiple faults, the performance of O is expected to degrade considerably, as it assumes that one component must be responsible for all failing tests [Abreu et al., 2009c]. For this reason, previous research has focused on multiple-faulted scenarios by either proposing less optimal fault predictors whose performance does not degrade as severely in the presence of multiple faults (*e.g.*, the D* predictor [Wong et al., 2014]); and by proposing more intricate techniques such as reasoning-based *SFL* (Section 1.2.2), that are not only more computationally expensive but also may increase the complexity of the debugging process.

However, what remains to be seen is how often such multiple-bug scenarios actually happen in practice. Our hypothesis is that, more often than not, programmers detect and fix *one bug at a time* during development. This would mean that, most often, developers are faced with single-faulted scenarios, so the use of the optimal fault predictor O could be justified given the prevalence of single-faults—and thus we would be able to not only provide optimal diagnostic reports to developers but also also make better use of tools and techniques that take diagnostic reports as input, such as with program repair [Nguyen et al., 2013]. Note that our hypothesis does not state that the system is only single-faulted at any particular point in time, but rather that faults are mostly detected in isolation.

To assess that such single-fault fix prevalence actually exists in practice, we describe a methodology that mines a project's code repository to find bug fixes and label them as being single- or multiple-faulted. The repository miner performs a reverse-chronological exploration of commits and runs newer test suites against older versions of the program. If a passing test-suite fails against an older codebase, it means that the code changes between the two versions (*i.e.*, Δ) contain a fix. If, on the other hand, there are compilation or runtime errors while running tests (due to, *e.g.*, a change in the interface between components), then we consider Δ as adding new functionality—so there is no fix present. Our classifier will then find if there is any component in Δ that appears in every affected test. If so, the fix is considered to be single-faulted. Otherwise, the fix will be labeled as multiple-faulted. Our methodology is similar to those of Böhme and Roychoudhury [2014], Dallmeier and Zimmermann [2007], and Sliwerski et al. [2005], in that code repositories are explored to isolate fixes.

We conducted a large-scale empirical study where we analyzed the repositories of 279 real, open-sourced Java projects, cataloged every detected fix, and performed fault-localization using the 5 popular predictors described in Section 1.2.1. In total, 1375 fixes were found. Out of all fixes, 1135 of them were single-faulted, thus yielding a prevalence of 82.5%. Among single-faulted fixes we observed that the O predictor has the best accuracy out of the tested predictors, with the faulted component being placed at the top of the diagnostic report in over 90% of all cases. Additionally, we found that another predictor proposed in the literature (O^P, a non-optimal variant of O [Naish et al., 2011]) performed similarly to O, while other predictors were less accurate. For multiple-faulted fixes, the diagnostic performance of O decreased considerably, making its fault localization reports unsuited for analysis. Other predictors showed a less severe performance degradation.

After analyzing the results, we have verified our hypothesis that most failures developers face are due to only one (active) bug, as there is a prevalence of single-fault fixes. However, our results suggest that the optimal O predictor's accuracy deteriorates significantly in the presence of multiple faults. On the upside, the O^P fault predictor has shown comparable performance to the optimal O in the case of single-faults, while still producing usable results for diagnosing multiple-faults.

This chapter's contributions are:

- A methodology for finding fixes in a software repository and labeling them as single- or multiple-faulted.
- Empirical evidence that single-faulted fixes correspond to 82.5% of all fixes in open-source Java projects.
- An assessment of the diagnostic performance of spectrum-based fault predictors in single-faulted scenarios. The optional O predictor, as well as O^P, show a degree of accuracy (with virtually no wasted effort) when compared to other predictors.
- An assessment of diagnostic performance in multiple-fault scenarios. We show that O's performance is essentially random. For other predictors, there is still a performance decrease, not as significant as O's, especially when trying to find the last faulty component in the ranking.

3.2 Cardinality of Fixes

Throughout this chapter, we use the terms *single-fault fix* and *multiple-fault fix* as defined in Section 1.3.2 (Definitions 7 and 8, respectively).

To identify a fix as single-faulted, we check if all tests affected by the change—*i.e.*, that went from failing to passing—share at least one component modified by the fix. For that, we look at the minimal-cardinality *hitting-set* (or *MHS*) of tests affected by the change. If there are hitting sets of cardinality 1, it means that at least one of the components modified by the fix is active on every affected test. Multiple-fault fixes are ones whose *hitting-sets* are of size greater than one, in which more than one component necessarily have to behave abnormally to explain the set of faulty test outcomes.

3.3 Methodology for Fault Classification

This section details the methodology we followed for mining a project's repository, finding fixing commits, and labeling them as either single- or multiple-fault fixes. A diagram depicting the methodology is shown in Figure 3.1. It should be noted that, although we motivate our approach by mentioning *SFL*, the methodology described in this section is completely separate from any diagnostic process.

3.3.1 Mining Fixing Commits

We employ a methodology for fault classification that involves access to the subject program's code repository, to enable the inspection of both the project's commit history and each *commit tree* (which represents the state of all checked-in files

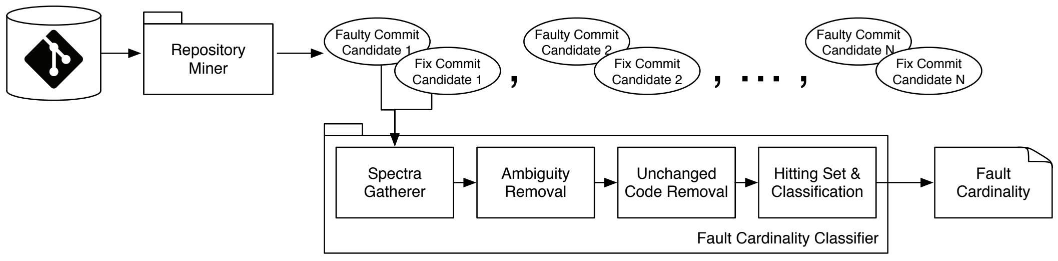


Figure 3.1: Methodology for mining and classifying fixes.

at a particular commit). We start by analyzing the latest commit in the main branch—which, in most workflows, is the `master` branch—and iteratively explore parent commits. This reverse-chronological exploration is able to handle most workflows enabled by modern version control systems (*e.g.*, `git`), such as branch merging, *rebasing* and commit *cherry-picking*[1]. However, note that the use of advanced history-rewrite features—like *commit squashing*[2]—may influence the outcome of the fault cardinality classifier, as these features allow sets of commits to be deleted, reordered, and even collapsed into one.

During our reverse-chronological analysis, we restore the working tree of the commit currently being explored and run the project's test-suite. If it is a passing suite, the commit is then considered as a fix candidate, and we advance to its parent commit, restoring its working tree. After that we run the fix candidate's suite—hence why our analysis is reverse chronological: so that the fixing commit's test-suite is run against an earlier commit. If the suite fails, we prompt the fault cardinality classifier, described in Section 3.3.2, to run. If, on the other hand, the suite passes, the commit's own test-suite is run to decide whether it should be the new fix candidate. This process repeats until all commits are explored.

3.3.2 Classifying Fault Cardinality of a Fixing Commit

We now describe the methodology for classifying the fault cardinality of any fixing commit discovered in Section 3.3.1. At this stage we execute test-suites at each commit under analysis and perform code coverage instrumentation. We have selected method-level granularity for the instrumentation so that methods are the units of our analysis. This way, fixes that only involve one method are classified as single-faulted. Our classification methodology encompasses four steps:

Gathering Spectra The first step in the fault classification process is to run the fixed version's test-suite against both fixed and faulty programs and gather their spectra with methods as the component granularity. By faulty programs we mean programs compiled from source code in which the fixing set of commits was rolled back. When

[1]*Cherry-picking* refers to the act of applying the changes from a set of commits to the current branch.
[2]*Commit squashing* is the act of merging together a series of commits so they appear as one in the commit history.

testing the suite against the fixed version, we ensure that every test is passing. Since tests pass in the fixed version, we attribute any test failure observed when testing the faulty program to the code changes between the two versions under analysis.

Figure 3.2 depicts example spectra generated by the two test runs. Figure 3.2a shows the faulty version's spectrum and Figure 3.2b shows the fixed version's counterpart. Highlighted components denote elements from the Δ set—the set of components that were modified between the two versions under analysis.

	t_1	t_2	t_3	t_4	
c_1	1	0	0	1	Δ
c_2	1	1	0	0	
c_3	0	1	0	1	Δ
c_4	0	0	1	0	
c_5	1	1	0	0	
c_6	0	1	0	1	
c_7	0	0	1	0	
e	0	1	0	1	

(a) Faulty version.

	t_1	t_2	t_3	t_4	
c_1	1	0	1	1	Δ
c_2	1	1	0	0	
c_3	1	1	0	1	Δ
c_4	0	0	1	0	
c_5	1	1	0	0	
c_6	0	1	0	1	
c_7	0	0	1	0	
e	0	0	0	0	

(b) Fixed version.

Figure 3.2: Spectra gathered when running test-suite from the fixed version. Δ denotes components changed by the fixing commit.

Note that, for the suite to run against the faulty version, the Δ set must not include any interface changes that render the two versions incompatible. Consequently, if any compilation or runtime errors arise while attempting to run the test suite, Δ is considered as containing changes to functionality (rather than a fix) and is therefore discarded from subsequent analysis.

Ambiguity Removal After gathering the faulty version of a spectrum, we perform an initial filtering step to remove ambiguous components, so that only one component from an ambiguity group is present. At the spectrum level of abstraction, components can form an ambiguity group (also known as an equivalence class) if they always exhibit the same execution behavior, so it is not possible to distinguish between them, as discussed in Section 2.3.3. This inter-dependence means that these components will always need to be inspected together. Therefore, if a bug occurs in an ambiguity group, the group will be considered as faulty. An example of this filtering step is depicted in Figure 3.3. We consider that if any component in ambiguity group belongs to Δ, then the ambiguity group also belongs to Δ. This is so that the newly created ambiguity group component can be considered in the next steps of the analysis.

Unchanged Code Removal The faulty version's spectrum shows test failures not present in the fixed spectrum (*cf.* Figure 3.2). Recall that the test-suite is unchanged between the two versions, as described in our step 1, so we can attribute the cause

	t_1	t_2	t_3	t_4	
c_1	1	0	0	1	Δ
c_2	1	1	0	0	
c_3	0	1	0	1	Δ
c_4	0	0	1	0	
e	0	1	0	1	

Figure 3.3: Ambiguity group filtering step. Components from Figure 3.2a that exhibit the same behavior are grouped and collapsed into a single component.

of the erroneous behavior to a subset of components $C \subseteq \Delta$ which is part of the code modified between the two versions under test[3]. All components not in Δ can therefore be safely exonerated from suspicion of containing the observed fault and are filtered out from the analysis, as shown in Figure 3.4.

	t_1	t_2	t_3	t_4
c_1	1	0	0	1
c_3	0	1	0	1
e	0	1	0	1

Figure 3.4: Filtered from Figure 3.3 components not involved in Δ.

Hitting Set & Classification The last filtering step to be performed is one that looks at failing tests from the faulty spectrum—namely t_2 and t_4 from our example—and keeps them in the analysis. Passing tests—t_1 and t_3—are discarded, as they do not reveal information about the faulty components.

The final, filtered spectrum, shown in Figure 3.5, is then submitted to *MHS* analysis [Abreu and van Gemund, 2009; Feldman et al., 2008] so that we are able to determine what (sets of) components are active on every failing row of the spectrum. Akin to what happens in reasoning-based *SFL* (*cf*. Section 1.2.2), these sets of components can be regarded as diagnostic candidates, since, when assumed faulty, can explain every observed error in the system. Each diagnosis candidate—a subset of all components—is valid if every failing test-case involves at least one component from the candidate. A candidate is *minimal* if removing any component from it makes it no longer a hitting set. We are only interested in minimal candidates, as they can subsume others of higher cardinality.

	t_2	t_4
c_1	0	1
c_3	1	1
e	1	1

Figure 3.5: Spectrum depicting a single-fault after filtering passing tests from Figure 3.4.

[3]This assumes test outcomes are deterministic.

If the minimal hitting set of the filtered spectrum yields solutions of cardinality 1, it means that there is at least one component that is involved every fault-revealing test. In Figure 3.5, component c_3 is active in every test, so we consider the fixing commit as being a *single-fault fix*.

Another example is depicted in Figure 3.6. In this filtered spectrum, the set of components $\{c_9, c_{10}\}$ has the minimal cardinality so that it explains every test failure. The fact that this spectrum's hitting set contains candidates of cardinality greater than 1 means that there is no one component that, when modified, causes all tests to pass. This is true because no component is active on every failing test. Therefore, in this case, the fixing commit is labeled as a *multiple-fault fix*.

	t_5	t_6	t_7
c_8	0	1	0
c_9	1	1	0
c_{10}	0	0	1
e	1	1	1

Figure 3.6: Spectrum depicting a multiple-fault.

3.4 Evaluation

In Section 1.2, we described several similarity-based *SFL* predictors and stated that, as they yield a uni-dimensional ranking of fault likelihood, their accuracy might be impacted in multiple-fault scenarios. However, we hypothesize that such multiple-faulted scenarios are not that frequent during development because developers tend to fix faults soon after they are detected. We aim to assess the prevalence of single-fault fixes in several real, open-source software projects.

Furthermore, for cases where multiple-faults are present in a system, we aim to quantify what is the decrease in diagnostic performance (if any) of using such similarity-based *SFL* techniques to debug the system.

This work aims to address the following research questions:

Research Question 3.1

How prevalent are single-fault fixes in open-source projects?

Research Question 3.2

What is the effort to diagnose single-faults with *state-of-the-art* fault predictors?

Research Question 3.3

What is the impact on diagnostic performance when multiple-faults are considered?

RQ3.1 is concerned with the quantitative assessment of single-fault fixes and how their pervasiveness compares to that of multiple-fault fixes. In **RQ3.2**, we ask what is the diagnostic efficiency of current similarity-based approaches—most of which designed to pinpoint single-faults—when solely considering single-faulted scenarios. In **RQ3.3**, we shift our attention towards multiple-faulted scenarios, and ask what the diagnostic performance of similarity-based *SFL* techniques is for these scenarios, with the aim of comparing against the single-faulted baseline.

We conducted a large scale empirical study encompassing hundreds of open-source projects. The experiment entailed mining their code repositories and finding fault-fixing commits following the methodology described in Section 3.3. Afterward, each pinpointed fault was diagnosed using the fault-localization techniques described in Section 1.2.1.

3.4.1 Experimental Setup

The subjects of our study are open-source software projects originally gathered for a study on pull request distributed development on GitHub, conducted by Gousios and Zaidman [2014]. The catalog encompasses over 6,000 publicly available code repositories for projects written in Java, Javascript, Python, Ruby and Scala. We have chosen this dataset due to its breadth of subjects and the fact that the vast majority of them contain test-cases—which are a requirement for spectrum-based analyses. The dataset was, however, filtered to fit the needs of our experiment. We have applied the following filtering schemes:

1. We only consider the dataset's 1,288 Java projects. Projects written in other languages were discarded due to the fact that our tooling only handles Java source code (1,288 subjects out of 6,001).
2. Non-Apache Maven projects were discarded. Maven is a requisite for our analysis because we use one of its plugins to instrument code at runtime to obtain a test execution's program spectra. We also ensure that the `mvn compile` command terminates successfully and all dependencies can be resolved (701 subjects out of 1,288).
3. Projects should contain tests, otherwise fault-localization tools are unable to perform the analysis (279 subjects out of 701).

Out of all projects from the dataset, we end up considering 279 Java projects as our subjects[4]. On average, subjects' test-suites were comprised of 596 tests.

The repository miner for classifying fixes as described in Section 3.3 is available at `https://github.com/aperez/single-fault-prevalence`. The miner uses the Python library GitPython[5] to iterate throughout the repository's history. Gathering program spectra is done through a Maven plugin that shares the same internals for runtime instrumentation of Java programs as the GZOLTAR fault-localization tool [Campos et al., 2012]. The fault classifier's *MHS* computation is performed using Abreu and van Gemund [2009]'s STACCATO algorithm. Fault localization is performed at the method granularity.

3.4.2 Metrics Used

To assess diagnostic performance, we resort to the wasted effort measurement described in Section 1.2.3. In the case of multiple-fault scenarios, since all diagnostic candidates generated by similarity-based *SFL* are single components, an effort metric by itself is insufficient to judge diagnostic efficiency [Steimann et al., 2013]. This is because more than one faulty component is scattered throughout the ranked list produced by similarity-based techniques. Given a set of k faulty components $\mathcal{F} = \{f_1, f_2, \cdots, f_k\}$, to better assess the diagnostic efficiency in these scenarios, we provide three measurements:

1. First-fault effort, which is the effort required to reach the first faulty component:

$$\min \{\text{Effort}(f) | f \in \mathcal{F}\} \tag{3.1}$$

2. Average-fault effort, an average of efforts to reach all faulty components:

$$\overline{\{\text{Effort}(f) | f \in \mathcal{F}\}} \tag{3.2}$$

3. Worst-fault effort, the effort required to reach the last faulty component in the ranking:

$$\max \{\text{Effort}(f) | f \in \mathcal{F}\} \tag{3.3}$$

We also compare the performance of each multiple-fault scenario to an artificially-crafted single-faulted equivalent. The artificial scenario is a proxy for a spectrum that only contains one component responsible for all erroneous behavior, aiding us to compare and contrast the outcome of fault-localization techniques between single- and multiple-faulted versions of the same problem. For that, we merge

[4]The full list of experimental subjects is available at `https://github.com/aperez/single-fault-prevalence`.

[5]Available at `https://pypi.python.org/pypi/GitPython`.

all faulted components into one, as depicted in the example from Figures 3.7a and 3.7b. Figure 3.7a shows a program spectrum with its two faults highlighted—*i.e.*, components c_1 and c_3. Our merging strategy creates a new spectrum (Figure 3.7b) with all faulty components stripped, in which a new component with the faulty components' coverage activity is inserted—this essentially amounts to performing a *bitwise or* among all faulty rows from the original spectrum. This way, we can judge what is the impact on diagnostic accuracy by measuring:

$$\Delta_{\text{Effort}} = \text{Effort} - \text{Effort}_{\text{merged}} \tag{3.4}$$

Δ_{Effort} values range from -1 to 1. A value of 1 means that the diagnostic efficiency is minimal in the multiple-fault scenario and maximal in its single-faulted equivalent. Conversely, $\Delta_{\text{Effort}} =$ -1 states that efficiency is maximal for the original scenario and minimal for the single-faulted one. $\Delta_{\text{Effort}} = 0$ means that both scenarios yield the same effort to diagnose.

	t_1	t_2	t_3	t_4
c_1	0	1	0	0
c_2	1	0	1	0
c_3	1	0	0	1
c_4	0	0	1	0
e	0	1	0	1

(a) Before faulty component merger.

	t_1	t_2	t_3	t_4
c_2	1	0	1	0
c_4	0	0	1	0
$c_{1,3}$	1	1	0	1
e	0	1	0	1

(b) After faulty component merger.

Figure 3.7: Multiple-fault components merge strategy.

In the eventuality that the fault classifier's *MHS* set step produces more than one minimal-cardinality result, it means that the spectrum has more than one fault candidate—*i.e.*, there are multiple sets of components that can independently explain failing tests. Figure 3.8 provides an example scenario where the hitting set encompasses two fault candidates of cardinality 1: c_1 and c_3. At the spectrum level of abstraction, one cannot distinguish the real fault among the minimal-cardinality candidates. To do so, one has to look at the source code from the fixing commit that provides the ground truth. However, from an *SFL* perspective, the fact is that any fault candidate could contain the fault. Since we are interested in the general case, we average the fault predictor values and effort scores for every scenario that has more than one minimal-cardinality set able to explain all observed errors. Note that this applies to both single-faulted and multiple-faulted scenarios.

3.4.3 Results

Out of the 279 subjects considered for evaluation, our classifier found fixing commits in 72 of them. What this figure tells us is not that 207 projects did not have any bug fixes, but rather that test-suites, when run against older versions of the code,

	t_1	t_2	t_3	t_4
c_1	1	1	0	0
c_2	1	0	0	1
c_3	1	1	1	0
c_4	0	0	1	1
e	1	1	0	0

Figure 3.8: Spectrum with multiple minimal-cardinality hitting sets.

do not produce any test-failure (although some still produce runtime errors, which are discarded, following the methodology described in Section 3.3.2. Also regarding these 207 projects, we can state that when developers find bugs in the code, they either (1) do not create a test-case exercising such a fault or (2) do not isolate their changes, potentially adding code for new functionality along with fixes in the same commit causing it to no longer being labeled as a bug-fixing commit.

Overall, 12417 commits were inspected, resulting in 1375 detected fixes (11% of inspected commits). Out of the detected fixes, 1135 of them (82.5%) were single-faulted, where one of the components modified by the fixing commit is sufficient to explain the faulty version's failing tests.

The histogram and violin plot from Figure 3.9a illustrate single-fault prevalence on different projects. For example, the prevalence of single faults ranges from 80 to 90% for 15 projects. The vertical dashed line in the plot indicates the median value of single-fault prevalence, 91.1%. Note that more than half of all projects considered show single-fault prevalence of 90% or more. Even considering the 25% percentile, the prevalence figure amounts to 79.6%. These prevalence results attest to the ubiquity of single-faults in open-sourced projects. Figure 3.9b shows fault cardinality for all detected fixes. It shows the quantity of fixes decaying exponentially with the fault cardinality.

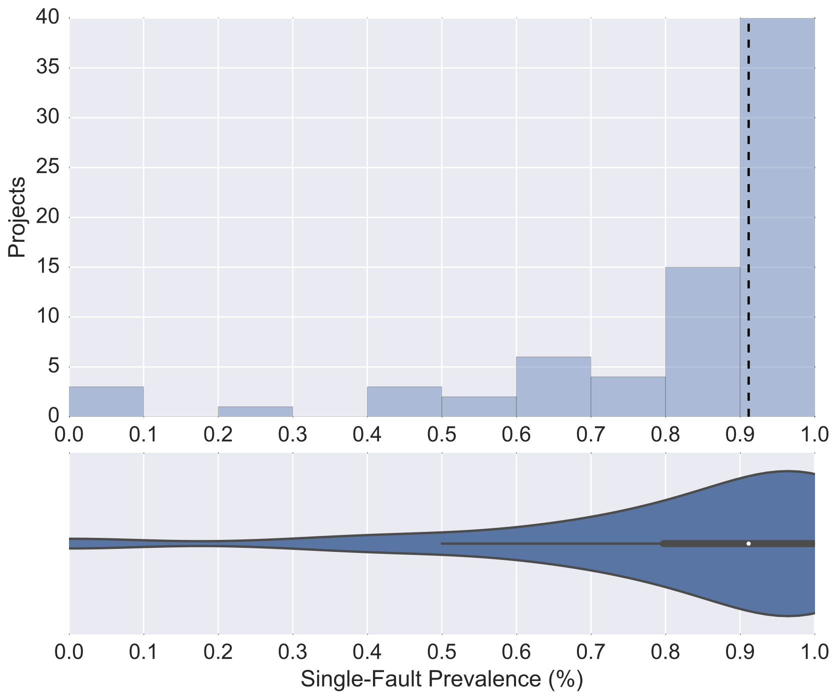
Revisiting the first research question:

Research Question 3.1

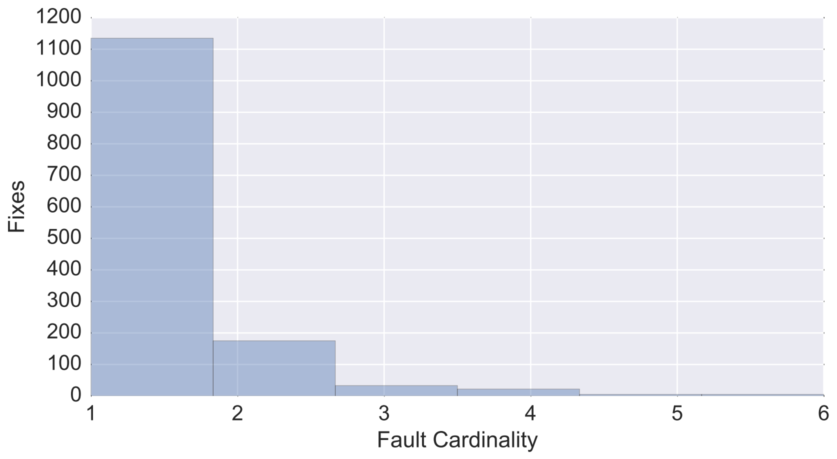
How prevalent are single-fault fixes in open-source projects?

Answer: We observed that 1135 fixes out of 1375 have eliminated a single fault from the system, yielding a single-fault prevalence of 82.5%.

We now shift our attention to the fixes classified as single-faulted to assess their diagnostic performance. For that, we show Figure 3.10, which is a cumulative plot of the effort required to detect the faults in our dataset when following the ranking generated by each fault predictor. The dashed vertical line represents an effort threshold of 0.05. We visually show this threshold because one should not assume developers continue following the fault-localization ranking at this point, particularly



(a) Histogram and violin plot of single-fault prevalence. A violin plot is the combination of a box plot and a kernel density plot.



(b) Histogram of fault cardinality.

Figure 3.9: Quantitative analysis of detected fixes.

when considering large codebases. This thresholding criterion is in agreement with the study by Parnin and Orso [2011], where developers were found to abandon the ranking if they inspected too many false positives; and the work of Nguyen et al. [2013] in program repair, that uses the predictor ranking as a set of clues to start the automated repair process, and places an explicit time-bounded threshold in the exploration of the ranking.

It is immediately apparent that the O fault predictor exhibits the highest diagnostic efficiency, with over 90% of faults being at the top of their respective rankings (since

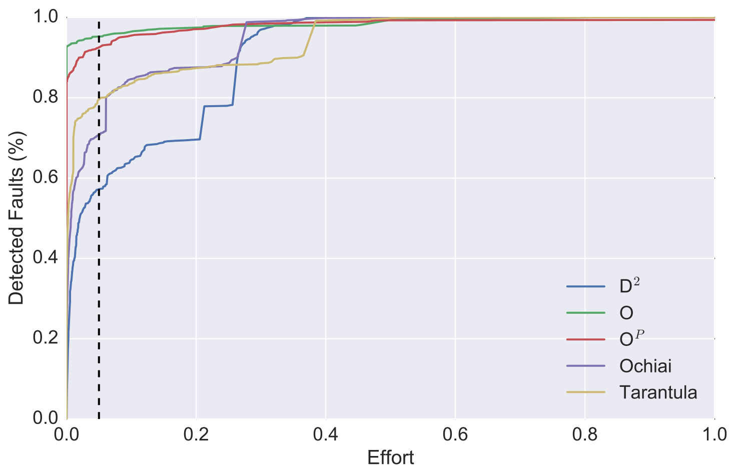


Figure 3.10: Diagnostic effort throughout single-fault scenarios. Values at threshold: D^2: 57.2%, **O: 95.2%**, O^P: 92.5%, Ochiai: 70.6%, Tarantula: 79.5%

their low effort value). O^P also fares comparably to O, diagnosing over 80% of faults with virtually zero wasted effort. In fact, at the exploration threshold, O and O^P manage to detect 95.2% and 92.5% respectively, attesting to their accuracy. Other fault predictors fare worse compared to both O and O^P. In fact, at the effort $= 0.05$ threshold highlighted in the vertical dashed line, other fault predictors' detection rate ranges from 57.2% to 79.5% of total faults.

Revisiting the second research question:

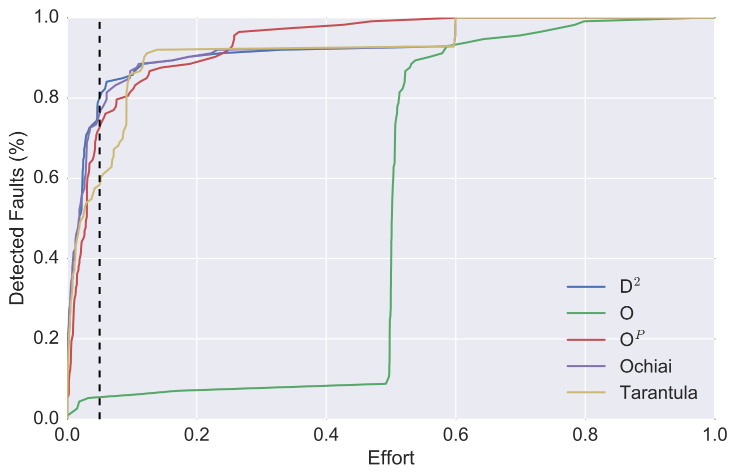
Research Question 3.2

What is the effort to diagnose single-faults with *state-of-the-art* fault predictors?

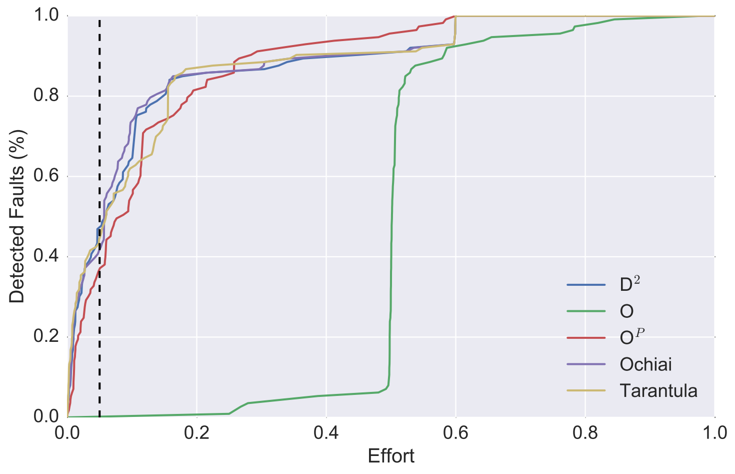
Answer: We show diagnostic for several fault predictors. One that fared consistently well was the O metric, with a fault detection rate amounting to over 90% while having virtually zero wasted effort.

Lastly, we look at multiple-fault scenarios. It is worth reminding the reader that a single effort measurement is insufficient to accurately portray diagnostic efficiency for multiple-faulted spectra, since we are using similarity-based *SFL* to diagnose. The effort metric measures the number of inspections required until the fault is reached. However, the erroneous behavior spans more than one component in these multiple-faulted scenarios. Hence, we introduce in Section 3.4.1 the notion of *first-fault*, *average-fault* and *worst-fault* efforts. Figures 3.11a to 3.11c plot such metrics for multiple-faulted scenarios.

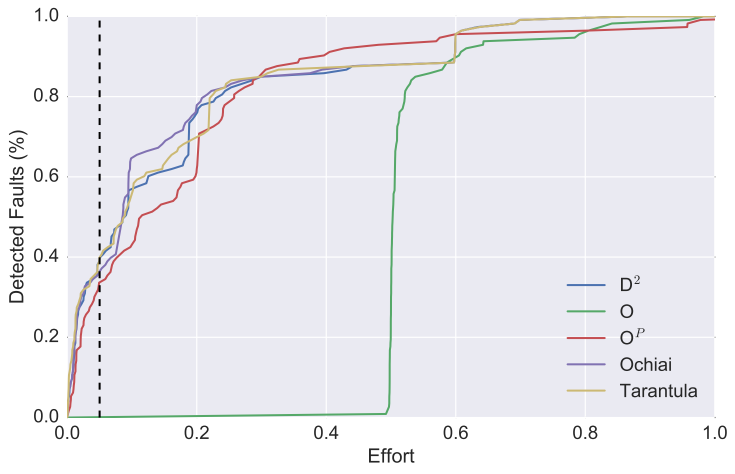
From the outset, we can notice that first-fault, average-fault and worst-fault efforts for the O fault predictor are very different from the remaining predictors. The predictor detects a very low amount of faults at low effort, and exhibits a sudden



(a) First-fault measurement in multiple-fault scenario. Values at threshold: **D^2: 79.6%**, O: 3.5%, O^P: 72.6%, Ochiai: 75.2%, Tarantula: 58.4%.



(b) Average-fault measurement in multiple-fault scenario. Values at threshold: **D^2: 46.9%**, O: 0%, O^P: 36.3%, Ochiai: 41.5%, Tarantula: 44.2%.



(c) Worst-fault measurement in multiple-fault scenario. Values at threshold: **D^2: 39.8%**, O: 0%, O^P: 33.6%, Ochiai: 36.2%, Tarantula: 39.5%.

Figure 3.11: Diagnostic effort throughout multiple-fault scenarios.

jump in detection rate at effort's halfway point. In fact, this is to be expected, since according to O's definition, it attributes a score of -1 to every component in which $n_{01} > 0$. In other words, any component that is not active in every failing test, is given a negative score. However, in multiple-fault scenarios, there is rarely ever a faulty component active in all failing tests, meaning that most components in the system will be scored with the same value, at which point locating the fault becomes essentially a random process. The other predictors produce fairly high fault-detection rates at low effort values to reach at least one of the faulty components (*i.e.*, considering the first-fault effort), with D^2, O^P and Ochiai exhibiting over 70% detection rate at the effort $= 0.05$ threshold. As to be expected, the detection rate decreases from the first-fault to the average-fault and from the average-fault to the worst-fault. Considering worst-fault effort (*i.e.*, the effort required to pinpoint all faults in the system), detection rates range from 33.6% using O^P to 39.8% using D^2.

Table 3.1 provides some additional information about the multiple-faulted scenarios. The table shows, for each fault predictor, its median value, median effort, and median Δ_{Effort}—used for comparing against an equivalent single-faulted scenario, as described in Section 3.4.1. A statistical test we performed was the was the Shapiro-Wilk test for normality of effort data [Shapiro and Wilk, 1965]. The results tell us that the distributions are not normal, with confidence of 99%. Given that the effort data is not normally distributed and that each observation is paired, we use the non-parametrical statistical hypothesis test Wilcoxon signed-rank [Wilcoxon, 1945]. Our null-hypothesis is that the median difference between the two observations (*i.e.*, Δ_{Effort}) is zero. We show the resulting Z statistic and p-value in Table 3.1. With 99% confidence, we can refute the null-hypothesis in all scenarios except for D^2 first-fault, Tarantula first-fault, Ochiai average-fault and Tarantula average-fault. In these cases, the effort values are comparable to their single-faulted counterparts. In cases where the null-hypothesis is refuted, only one yielded a negative Δ_{Effort} — Ochiai first-fault — meaning that finding the first component out of the multiple components that comprise the fault was faster than finding the merged single-faulted component. All in all, we can say that, except for the O metric where Δ_{Effort} has a large magnitude, the effort measurements are comparable to their single-faulted counterparts when we consider the effort required to find one fault in the ranking. Diagnostic performance decreases when considering the effort required to find all faults in the system.

Revisiting the third research question:

Research Question 3.3

What is the impact on diagnostic performance when multiple-faults are considered?

Table 3.1: Metric medians and statistical tests in multiple-faulted scenarios.

		Median Value	Median Effort	Median Δ_{Effort}	Wilcoxon Signed-rank
First Fault	**D**^{2}	1.00	0.02	0.00	$Z = 3059.0$ p-value $= 0.885$
	O	-1.00	0.50	0.50	$Z = 0.0$ p-value $= 4.07\times10^{-20}$
	O^P	3.01	0.03	0.02	$Z = 604.5$ p-value $= 2.85\times10^{-13}$
	Ochiai	0.56	0.02	-0.02	$Z = 1046.0$ p-value $= 6.06\times10^{-09}$
	Tarantula	0.99	0.02	-0.02	$Z = 2351.0$ p-value $= 3.64\times10^{-02}$
Average-Fault	**D**^{2}	0.96	0.05	0.04	$Z = 2183.0$ p-value $= 2.95\times10^{-03}$
	O	-1.00	0.50	0.50	$Z = 0.0$ p-value $= 2.78\times10^{-20}$
	O^P	2.50	0.07	0.07	$Z = 0.0$ p-value $= 2.78\times10^{-20}$
	Ochiai	0.42	0.05	0.00	$Z = 2901.0$ p-value $= 0.651$
	Tarantula	0.92	0.05	0.00	$Z = 2656.0$ p-value $= 0.105$
Worst-Fault	**D**^{2}	0.17	0.08	0.06	$Z = 2070.0$ p-value $= 9.79\times10^{-04}$
	O	-1.00	0.50	0.50	$Z = 0.0$ p-value $= 2.78\times10^{-20}$
	O^P	1.03	0.10	0.09	$Z = 0.0$ p-value $= 2.78\times10^{-20}$
	Ochiai	0.26	0.07	0.03	$Z = 1914.0$ p-value $= 1.05\times10^{-03}$
	Tarantula	0.85	0.07	0.03	$Z = 1950.0$ p-value $= 2.72\times10^{-04}$

Answer: With the exception of the O fault predictor, which performs with random accuracy, the first-fault effort measurements of other fault predictors are comparable to the diagnostic effort for single-faulted equivalent scenarios. To diagnose all faults in a system, the fault predictors' accuracy decreases. Aside from O, the performance among other predictors when faced with multiple-fault scenarios is similar.

3.4.4 Threats to Validity

Potential threats to the validity of our experiment are the following:

External Validity When choosing the projects for our study, our aim was to opt for projects that resemble general, large-sized application being worked on by several people. To reduce selection bias and facilitate the comparison of our results, we decided to use the real-world subjects collected in the dataset gathered by Gousios and Zaidman [2014].

Construct Validity A potential threat to construct validity relates to our definition of what constitutes single-faulted and multiple-faulted fixes (*cf*. Section 3.2). Additionally, another threat to construct validity is our assumption that any interface change is result of a change in requirements and not the consequence of a fix. Lastly, we point out that history-rewrite features of modern version control systems can influence the outcome of the fault cardinality classifier. It can be the case that many single-faulted fixing commits collapsed into one large commit that is responsible for fixing multiple-faults by means of *commit squashing*.

Internal Validity The main threat to internal validity lies in the complexity of several of the tools used in our experiments, most notably our code instrumentation tool to retrieve spectrum information.

3.5 Discussion

We list below some of the practical implications of this study:

- We argue that our experimental results suggest a methodology to be followed when developers face failing test cases. As we have shown that there is a high likelihood that there is only one bug detected by failing tests, developers may try to find the fault by inspecting the ranking generated by the O^P fault predictor, since it produces near-optimal scores in the event of single-faults while still being usable in multiple-faulted scenarios (unlike the O predictor).
- Results suggest that closely monitoring the system as it develops (through, for instance, a continuous integration platform) and attempting to locate faults as soon as failures emerge will yield debugging tasks that require less wasted effort. This is because the likelihood of the fix being single-faulted is high when compared to only dealing with debugging tasks once there is a significant number of failing tests.
- Further research is needed in order to find whether there is a fault predictor that is closer to showing optimal accuracy when diagnosing multiple faults, exhibiting a Δ_{Effort} that approaches zero for worst-fault scenarios.
- Effective automatic fault localization paves the way to other automatic techniques, such as automated program repair. Our experimental results yields insight into which technique will work best in practice. In particular, the prevalence of single-fault fixes suggest that the O^P fault predictor will yield near-optimal rankings as input to automatic repair techniques in the event

of single faults, while still providing some guidance in the event of multiple faults.

3.6 Related Work

There is a large and varied body of work on software repository mining approaches, following the general theme of further understanding the practice of making software [Herzig and Zeller, 2010]. These mining approaches cover topics such as identifying change patterns [Gall et al., 1998]; predicting defects [Bettenburg et al., 2012; F. Zhang et al., 2016] and vulnerabilities [Munaiah et al., 2017]; detecting functional clones [Rahman et al., 2012] and cross-project reuse [Gharehyazie et al., 2017]; and even identifying developers [Robles and González-Barahona, 2005]. Some fault localization approaches that use concepts from repository mining have been proposed, such as one by Cardoso and Abreu [2013a] which created a behavioral model of previous diagnoses to better estimate the component goodness parameter in reasoning-based *SFL*; one by Elmishali et al. [2016] that leveraged historical information from the project's versioning system and bug tracker; and one by Sohn and Yoo [2017] which leveraged source code metrics such as code size, age and code churn to rank diagnostic candidates.

Software fix mining (along with defect prediction) approaches—such as the ones proposed by Böhme and Roychoudhury [2014], Dallmeier and Zimmermann [2007], and Sliwerski et al. [2005]—share many similarities with our study in terms of methodology for gathering and analyzing bug fixes. One difference is that our methodology does not rely on syntactic analysis of commit metadata (such as, *e.g.*, looking for a bug tracking identification number in the commit message) like some related work. A potential downside of overlooking such metadata is that commits that not only fix bugs but also change functionality may be discarded from analysis (due to compilation/runtime errors while attempting to run a given test suite on an older version of the codebase). A potential upside is that our methodology does not assume the target project relies on a bug tracker.

As for studying the influence of multiple faults in *SFL* techniques, DiGiuseppe and Jones [2011], found that at least some kinds of faults were *localizable* regardless of the presence of other faults. The authors showed that, while the presence of more than one fault added noise to the ranking, such noise did not adversely affect the localizability of certain types of faults. To mitigate the influence of multiple faults, *debugging in parallel* approaches were also proposed [Jones et al., 2007; Steimann and Frenkel, 2012; Hogerle et al., 2014]. These approaches cluster test cases in order to partition the program spectrum into multiple single-faulted spectra, which can then be diagnosed independently. However, as our empirical results demonstrate, diagnosing single faults is still the more prevalent task. As such, we argue that *debugging in parallel* actually happens organically over the course of software development, as bugs are detected and fixed in isolation, with all other potential faults in the system remaining undetected.

3.7 Summary

In this chapter, motivated by the computational simplicity of similarity-based *SFL* (outlined in Section 1.3.2), we thoroughly measured the cardinality of fixes across a multitude of open-sourced software projects and analyzed the accuracy of similarity-based techniques at locating said fixes. Concretely, in this chapter:

- We describe a methodology for mining fixing commits in software repositories and for categorizing them according to the number of faults they address (Section 3.3, page 49). This approach can therefore provide an insight into how programmers find and fix bugs in practice.
- We conducted an empirical evaluation which mined fixing commits in 279 Java projects hosted on GitHub and analyzed fault-fixing behavior (Section 3.4, page 53). Results show that:
    1. Single-fault fixes account for 82.5% of the total 1375 fixing commits found (answers **RQ3.1**, page 57);
    2. Fixes catalogued as single faults generally exhibit low effort to diagnose across all similarity-based fault predictors used in the evaluation. The O predictor—shown to be optimal in single-faulted scenarios—fared consistently well across all projects, over 90% of which having virtually zero wasted effort (answering **RQ3.2**, page 59);
    3. The O predictor's accuracy severely decreases in the event of multiple faults. Other fault predictors are still sufficiently accurate at diagnosing at least one of the faults, but performance decreases when attempting to locate all faults (answering **RQ3.3**, page 62);.

A Qualitative Reasoning Extension to *SFL* 4

Poster: A Qualitative Reasoning Approach to Spectrum-based Fault Localization
Alexandre Perez and Rui Abreu
In: Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 3, 2018.

Leveraging Qualitative Reasoning to Improve SFL
Alexandre Perez and Rui Abreu
In: Proceedings of the 27th International Joint Conference on Artificial Intelligence and the 23rd European Conference on Artificial Intelligence, IJCAI-ECAI 2018, Stockholm, Sweden, July 13-19, 2018.

Abstract *SFL correlates a system's components with observed failures. By reasoning about coverage, SFL allows for a lightweight way of pinpointing faults. This abstraction comes at the cost of missing certain faults—such as errors of omission—and failing to provide enough contextual information to explain why components are considered suspicious. We propose an approach, named Q-SFL, that leverages Qualitative Reasoning to augment the information made available to SFL techniques. It qualitatively partitions system components, and treats each qualitative state as a new SFL component to be used when diagnosing. Our empirical evaluation shows that augmenting SFL with qualitative components can improve diagnostic accuracy in 54% of the considered real-world subjects.*

4.1 Introduction

Despite the developments and achievements in *SFL* research, we are unable to find many accounts of successful transitions of this technology into the software industry at large. We argue that this is motivated largely by the issues raised by Parnin and Orso [2011] in their user study of automated debugging techniques, corroborated by Steimann et al. [2013] and Pearson et al. [2017]. Namely, the authors found that there is significant interest drop-off after developers inspect a small number of components from the ranked list of potential faults. This issue is exacerbated as the scale of the system under test increases. Another issue pointed out by Parnin and Orso is the fact that many studies assume *perfect bug understanding*—that is, these studies expect that once developers inspect a faulty component, they will correctly identify it as such—, which does not always hold in practice [Gouveia et al., 2013]. Therefore, as argued in Section 1.3.3, an important goal is to find ways of *enriching* the diagnostic report generated by *SFL* techniques, to provide more information about *why* each highly-ranked software component is under suspicion.

This chapter proposes a significant departure from current efforts as we leverage the inspection of the runtime value for relevant variables and parameters from the system under test, with the intent of augmenting reports generated by *SFL* techniques and providing more diagnostic information to developers. Recording every instance of these units of data for each test quickly becomes intractable, even for a lightweight approach as *SFL*. Therefore, we leverage an Artificial Intelligence concept used for modeling and simulation of complex physical systems: **Qualitative Reasoning (*QR*)**. *QR* provides a way of describing continuous values by their discrete, behavioral qualities, to enable the ability of reasoning about a system's behavior without exact quantitative information [Forbus, 1997; Williams and de Kleer, 1991]. Precise numerical quantities are avoided and replaced by qualitative descriptions—such as, for instance: *high*, *low*, *zero*, *increasing* or *decreasing*.

We apply *QR* to the *SFL* analysis, in an approach named **Qualitative Spectrum-based Fault Localization (*Q-SFL*)**, enabling the introduction—both manually and automatically—of quantitative landmarks that will partition the domains of relevant data units into a set of qualitative descriptions, and insert a new *SFL* component for each of these descriptions. Since behavioral qualities are considered as components, we are then able to record their coverage for each test case and rank them according to their correlation to failing test runs, thus not only suggesting the likely location of the bug, but also pinpointing which behavioral properties induce a failure, *enriching* the *SFL* report as a result. This can have benefits in *bug comprehension*—as an example, a *Q-SFL* diagnostic report is able to tell users that a method is likely to exhibit faulty behavior when one of its parameters has a negative value—and even improve diagnostic report accuracy—whenever a qualitative state is more correlated with failing behavior than its enclosing faulty component.

We perform an empirical evaluation of *Q-SFL* with real-world faults from the DEFECTS4J catalog. Results show that *Q-SFL* has the potential to improve the accuracy of *SFL* reports—with 54% of considered subjects exhibiting a lower effort to diagnose faults. Although the results are promising, we discuss several matters that need further research—namely, uncovering a landmarking strategy that exhibits consistently better results and studying to what extent is the *bug comprehension* improved.

This chapter's contributions are:

- An approach, named *Q-SFL*, inspired by *QR* research, to augment program spectra used by *SFL* techniques by partitioning a variable's runtime value into a set of qualitative states which are treated as *SFL* components.

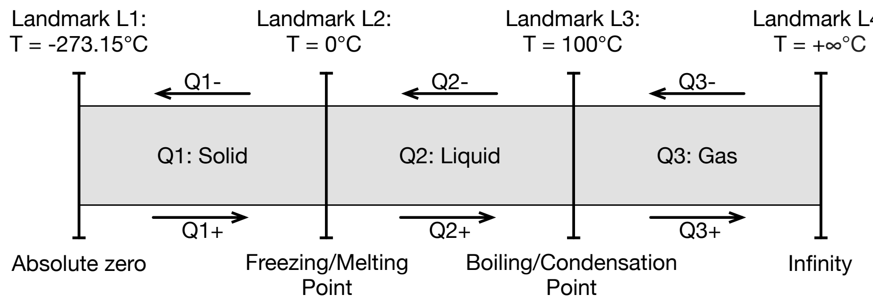


Figure 4.1: Example of a possible qualitative discretization of water temperature.

- Empirical evidence that *QR*-augmented spectra *can* reduce the effort to diagnose real bugs.
- A discussion on the practical implications of the approach and an outline of future research.

4.2 Qualitative Reasoning (*QR*)

QR is a field of Artificial Intelligence that creates a discrete representation of the continuous world [Forbus, 1997; Williams and de Kleer, 1991; de Kleer, 1977]. This enables the reasoning of space, time, and quantity with merely a small amount of information. It is motivated by the fact that humans are able to draw conclusions about the physical world around them with limited information, without the need of solving complex differential equations. [Forbus, 1997] also notes that advances in qualitative reasoning can help scientists and engineers—who appear to use qualitative reasoning when initially understanding a problem and when interpreting the results of quantitative simulations, calculations, and measurements.

Figure 4.1 provides an example of a potential discretization of the water temperature qualitative variable into three qualitative values: Q1, Q2 and Q3. Our representation *resolution*—*i.e.*, the granularity of the information detail—coincides with that of the three physical states of matter that water can assume: solid, liquid, and gas. Note that the established resolution will ultimately define the granularity of the conclusions one can draw from QR. To define the *qualitative states*, one needs to establish *landmarks*. Landmarks are constant quantitative values that establish a point of comparison to be able to reason about the qualitative states [Kuipers, 1986]. In our water temperature example, we know that if the water is in the liquid state (Q2), then its temperature is somewhere between landmark L2—corresponding to 0°C, the freezing point of water—and landmark L3—its boiling point, 100°C. Similarly, we can derive that ice (Q1) temperature assumes a value between the absolute zero (L1) and the melting point (L2); and that water vapor (Q3) ranges between the condensation point (L3) and positive infinity (L4). Depending on the use case and the domain of the variable under analysis, landmarks that coincide with domain limits—no values can be below L1 and above L4, in this case—may be disregarded, as they are not needed to infer a transition between states.

QR also supports the representation of *derivatives* between two quantities. They are usually represented with '+' and '-' signs, denoting value increases and decreases, respectively. This can enable, depending on the application, the use of sign algebra to reason about *direct influence* and *proportionality* between two qualitative values [Kuipers, 1986]. Derivatives also enable *envisionments*. An *envisionment* establishes a set of transitions between qualitative states [de Kleer, 1977], essentially modeling the abstracted world. A possible transition in our example's envisionment is the following: given that we observe Q2+—that is, we observe that the liquid water's temperature is rising—, then we know that the only possible following states are Q2 (continues in the liquid state) and Q3 (condensates into vapor), but never Q1 (freezes into ice).

In summary, with the *QR* framework, we establish a way to (i) represent quantities through discrete states, (ii) provide a way to compare values between these states, (iii) enable derivations and sign algebra, and (iv) model envisionments detailing possible transitions between states. With such a framework, we can model, plan, simulate and reason about a multitude of intricate problems in an abstract way.

4.3 Approach

As we outline in Section 1.3.3, *SFL* faces issues preventing it from widespread adoption. The abstracted nature of the coverage-based analysis, while lightweight, inevitably imposes accuracy and comprehension tradeoffs, may lead to the formation of *ambiguity groups*, and facilitates the occurrence of *coincidental correctness*.

We argue that the issues described above can be prevented, or at least attenuated, if we supplement the SFL framework with more contextual information about the system under analysis when diagnosing.

4.3.1 *Q-SFL*

Our *Q-SFL* approach consists of partitioning several *SFL* components into multiple, meaningful, qualitatively distinct *subcomponents*, to be used in the fault localization. We leverage the *QR* concept of domain partitioning to inspect existing components during each system execution and assign them a set of qualitative state. Each of these qualitative states is then considered as a separate *SFL* component whose involvement per transaction is recorded and fed into the *SFL* framework for diagnosis.

Figure 4.2 provides an example of how *QR* can be leveraged to enhance program spectra. Figure 4.2a shows a 4 transaction by 4 component spectrum, along with resulting diagnostic scores from applying reasoning-based *SFL*. Candidate generation

A	t_1	t_2	t_3	t_4
c_1	1	1	1	1
c_2	0	0	1	1
c_3	1	1	1	0
c_4	0	0	1	0
e	1	1	0	0

(a) Regular spectrum.
$\Pr(\{c_1\} \mid (A, e)) = 0.30$
$\Pr(\{c_3\} \mid (A, e)) = \mathbf{0.70}$

A'	t_1	t_2	t_3	t_4
c_1	1	1	1	1
c_1'	0	0	1	1
c_1''	1	1	0	0
c_2	0	0	1	1
c_3	1	1	1	0
c_3'	1	0	0	0
c_3''	0	1	1	0
c_4	0	0	1	0
e'	1	1	0	0

(b) *QR*-augmented spectrum.
$\Pr(\{c_1\} \mid (A', e')) = 0.05$
$\Pr(\{c_1''\} \mid (A', e')) = \mathbf{0.82}$
$\Pr(\{c_3\} \mid (A', e')) = 0.12$
$\Pr(\{c_3', c_3''\} \mid (A', e')) = 0.01$

Figure 4.2: Example of coverage partitioning via *QR*.

yields two candidate diagnoses—components c_1 and c_3 can independently explain the observed failures as both cover all failing test cases. For this example, suppose that c_1 is the faulty component. Since c_1 is involved in more passing transactions than c_3, while exhibiting the same behavior in faulty transactions, the *SFL* framework will assign it a lower fault probability than c_3—since c_3's activity pattern seems more correlated (*i.e.*, more similar) to the set of failing transaction outcomes. A likely explanation for such phenomenon is the fact that the faulty component may exhibit several distinct *modes of execution*—some of which may not trigger the fault—thus suitably explaining eventual ambiguity grouping or coincidental correctness phenomena. Hence, to improve the accuracy of the *SFL* framework, one needs more contextual information about component executions.

A prospective solution to the problem of encoding more contextual information in the spectrum is, then, to partition components as an attempt to capture their different execution modes. We envision three different types of *landmarking strategies* that can be employed to define qualitative state boundaries: (i) *manual landmarking*, where the system's developers manually define what are the possible qualitative states for a given component; (ii) *static landmarking*, where landmarks depend on the *type* of a component; and (iii) *dynamic landmarking*, where a component's value is inspected at runtime, and partitioned into a set of categories. Examples of dynamic strategies will be presented in Section 4.3.2 and section 4.4.

Figure 4.2b depicts the *QR*-augmented spectrum, where components representing qualitative partitions of both c_1 and c_3 are added to the original spectrum. An example of such partitioning using *static landmarking*: if c_1 represents a software procedure that contains a numeric parameter i, we can create two qualitative components c_1' and c_1'' that represent invocations of c_1 with $i \geq 0$ and $i < 0$, respectively.

`public double getMaximumExplodePercent() {`	t_1	t_2	t_3	t_4
Qualitative State: `this.dataset == null`				●
Qualitative State: `this.dataset != null`	●	●	●	
`+   if (this.dataset == null) {`				
`+     return 0.0;`				
`+   }`				
`double result = 0.0;`	●	●	●	●
`Iterator iterator = this.dataset.getKeys().iterator();`	●	●	●	●
`while (iterator.hasNext()) {`	●	●	●	
`  Comparable key = (Comparable) iterator.next();`	●	●		
`  Number explode = (Number) this.explodePercentages.get(key);`	●	●		
`  if (explode != null) {`	●	●		
`    result = Math.max(result, explode.doubleValue());`	●			
`  }`				
`}`				
`return result;`	●	●	●	
`}`				
Test outcome (e):	✓	✓	✓	✗

Figure 4.3: Chart 15 patch snippet with static landmarks.

This is a sign-based static partitioning strategy. Note that the original components c_1 and c_3 are not removed from the *QR*-augmented spectrum, as partitions may not provide further fault isolation. Since the original components remain in the spectra, they subsume unsuccessful partitionings which are uncorrelated with the failure, effectively yielding no increase in diagnostic effort.

If we are to diagnose the new spectrum from Figure 4.2b, component c_1'' is now the top-ranked diagnostic candidate. This *QR*-augmented spectrum avoids spurious inspections of component c_3, and provides additional *contextual information* about the fault, namely that $i < 0$ is often observed in failing transactions.

By landmarking data units associated with *SFL* components so that they are assigned a qualitative state at runtime, we are providing more context to the diagnostic process, and in some cases, consequently reducing the diagnostic effort. Such partitioning is also of crucial importance towards minimizing the impact and frequency of *ambiguity grouping* and *coincidental correctness*, as new, distinct components are added to the system's spectrum.

4.3.2 *Q-SFL* in Practice

To better depict the benefits of the *Q-SFL* approach, we walk through its application in real-world subjects from the DEFECTS4J collection of faults (introduced in Section 2.5.3). Figures 4.3 to 4.12 show the relevant code changes between faulty and fixed versions of a DEFECTS4J subject along with a potential landmarking scheme able to further isolate the fault. For brevity, this section details only a small subset of representative DEFECTS4J subjects.

	`public int getDomainAxisIndex(CategoryAxis axis) {`	t_1	t_2	t_3	t_4
	Qualitative State: `axis == null`			●	●
	Qualitative State: `axis != null`	●	●		
+	`  if (axis == null) {`				
+	`    throw new IllegalArgumentException("Null 'axis' argument.");`				
+	`  }`				
	`  return this.domainAxes.indexOf(axis);`	●	●	●	●
	`}`				
	Test outcome (e):	✓	✓	✗	✗

Figure 4.4: Chart 19 patch snippet with static landmarks.

	`public static Class<?>[] toClass(Object[] array) {`	t_1	t_2	t_3	t_4
	Qualitative State: `Arrays.asList(array).contains(null)`				●
	`  if (array == null) {`	●	●	●	●
	`    return null;`	●			
	`  } else if (array.length == 0) {`		●	●	●
	`    return ArrayUtils.EMPTY_CLASS_ARRAY;`		●		
	`  }`				
	`  Class<?>[] classes = new Class[array.length];`			●	●
	`  for (int i = 0; i < array.length; i++) {`			●	●
-	`    classes[i] = array[i].getClass();`			●	●
+	`    classes[i] = array[i] == null ? null : array[i].getClass();`				
	`  }`				
	`  return classes;`			●	
	`}`				
	Test outcome (e):	✓	✓	✓	✗

Figure 4.5: Lang 33 patch snippet with static landmarks.

	`public double getSumSquaredErrors() {`	t_1	t_2	t_3	t_4
-	`  return sumYY - sumXY * sumXY / sumXX;`	●	●	●	●
+	`  return Math.max(0d, sumYY - sumXY * sumXY / sumXX);`				
	Qualitative State: `_result == 0`			●	
	Qualitative State: `_result > 0`	●	●		
	Qualitative State: `_result < 0`				●
	`}`				
	Test outcome (e):	✓	✓	✓	✗

Figure 4.6: Math 105 patch snippet with static landmarks.

Static Landmarks

As mentioned in Section 4.3.1, static landmarking strategies are ones that can be applied by inspecting the code *locally*, without any specific knowledge about what the overall system is trying to achieve. Typical static analyses include type analyses and *def-use* analyses, which we also consider suitable for static landmarking. Figures 4.3 to 4.5 highlight potential uses of static landmarks able to create *SFL* components with higher fault correlation.

`private String getRemainingJSDocLine() {`	t_1	t_2	t_3	t_4
Qualitative State: `this.unreadToken == NO_UNREAD_TOKEN`	●	●		
Qualitative State: `this.unreadToken != NO_UNREAD_TOKEN`			●	●
`String result = stream.getRemainingJSDocLine();`	●	●	●	●
`+ this.unreadToken = NO_UNREAD_TOKEN;`				
`return result;`	●	●	●	●
`}`				
Test outcome (e):	✓	✓	✗	✗

Figure 4.7: Closure 133 patch snippet with dynamic landmarks.

Figure 4.3 depicts the patch applied to DEFECTS4J's Chart 15 bug. The bug lies in a method of the `PiePlot` class responsible for returning the maximum percentage of "exploded" slices—*i.e.*, the slices pulled apart from the chart. Field `this.dataset` is accessed without ensuring it is not `null`. This is an error of omission, and a *def-use* analysis allows us to generate the proper check `this.dataset == null`, which has a higher correlation with the error vector e than any other component.

Figure 4.4 depicts a similar nullity check omission error, but with a `null` argument instead of a `null` field. Interestingly, without qualitative components, this spectrum's single component is active on all test runs, exhibiting a form of *coincidental correctness*. Another example of *def-use* analysis is shown in Figure 4.5, where each element of an array-like argument is checked for nullity. The existence of a for loop invoking methods of each element in the collection justifies this static landmark, which manages to correlate with the fault.

Figure 4.6 exemplifies a sign-base check—as previously described in Section 4.3.1—for the value returned by the faulty method (labeled as `_result` in the figure). In this case, all activations of the qualitative state `_result < 0` will correlate with the failure, since the faulty method is intended to calculate the squared error—whose result is supposed to be non-negative (this implementation can return negative values due to rounding errors in the calculation of fields `sumXX`, `sumXY` and `sumYY`).

Dynamic Landmarks

With dynamic landmarking, variables' values are *probed* each run, with the aim of partitioning their domains into qualitative states, following some automated clustering criterion. One criterion might be, for instance, the euclidean distance between values, in the case of numeric variables. Partitionings can also be guided by test outcome, resembling a classification problem. Such inferred models can then be translated into qualitative state boundaries, so that augmented spectra can be generated.

`public void stop() {`	t_1	t_2	t_3	t_4
Qualitative State: `this.runningState == STATE_SUSPENDED`				●
Qualitative State: `this.runningState != STATE_SUSPENDED`	●	●	●	
`if(this.runningState != STATE_RUNNING && this.runningState != STATE_SUSPENDED) {`	●	●	●	●
`throw new IllegalStateException("Stopwatch is not running.  ");`	●	●		
`}`				
`+ if(this.runningState == STATE_RUNNING) {`				
`stopTime = System.currentTimeMillis();`			●	●
`+ }`				
`this.runningState = STATE_STOPPED;`			●	●
`}`				
Test outcome (e):	✓	✓	✓	✗

Figure 4.8: Lang 55 patch snippet with dynamic landmarks.

`public double doubleValue() {`	t_1	t_2	t_3	t_4
`double result = numerator.doubleValue() / denominator.doubleValue();`	●	●	●	●
`+ if (Double.isNaN(result)) {`				
`+ int shift = Math.max(numerator.bitLength(),`				
`+ denominator.bitLength()) - Double.MAX_EXPONENT;`				
`+ result = numerator.shiftRight(shift).doubleValue() /`				
`+ denominator.shiftRight(shift).doubleValue();`				
`+ }`				
`return result;`	●	●	●	●
Qualitative State: `Double.isNaN(_return)`				●
`}`				
Test outcome (e):	✓	✓	✓	✗

Figure 4.9: Math 36 patch snippet with dynamic landmarks.

The first example of dynamic landmarking is shown in Figure 4.7. In this example, the value of the field enumeration `this.unreadToken` was probed at every invocation of the method `getRemainingJSDocLine` and correlates with the failing test outcomes whenever its value is not `NO_UNREAD_TOKEN`—since, in fact, the state of `this.unreadToken` is supposed to be changed when invoking the faulty method. Given this, such qualitative partitioning can be considered in the *Q-SFL* analysis. Similarly, in Figure 4.8, the `stop` method of the `StopWatch` seems to fail whenever the field enumeration `this.runningState` is `STATE_SUSPENDED`, as the method tries to update the final stopping time of the stop watch, when the timer is already suspended. Figure 4.9 depicts the case where the test failures correlate with cases when the return value of the `doubleValue` method is `NaN`.

Manual Landmarks

A third category of landmarking strategies we envision having is manual landmarking. Manual landmarks correspond to cases where one cannot—at least in a general,

`private void removeUnreferencedFunctionArgs(Scope fnScope) {`	t_1	t_2	t_3	t_4
Qualitative State: `this.removeGlobals == false`				●
Qualitative State: `this.removeGlobals == true`	●	●	●	
`+   if (!this.removeGlobals) {`				
`+     return;`				
`+   }`				
`    Node function = fnScope.getRootNode();`	●	●	●	●
`    if (NodeUtil.isGetOrSetKey(function.getParent())) {`	●	●	●	●
`      return;`	●			
`    }`				
`    Node argList = getFunctionArgList(function);`		●	●	●
`    if (!modifyCallSites || !callSiteOptimizer.canModifyCallers(function)) {`		●	●	●
`      Node lastArg;`			●	●
`      while ((lastArg = argList.getLastChild()) != null) {`			●	●
`        if (!referenced.contains(fnScope.getVar(lastArg.getString()))) {`			●	●
`          argList.removeChild(lastArg);`			●	●
`          compiler.reportCodeChange();`			●	●
`        } else {`			●	●
`          break;`			●	●
`        }`				
`      }`				
`    } else {`		●		
`      callSiteOptimizer.optimize(fnScope, referenced);`		●		
`    }`				
`  }`				
Test outcome (e):	✓	✓	✓	✗

Figure 4.10: Closure 1 patch snippet with manual landmarks.

`public static float max(final float a, final float b) {`	t_1	t_2	t_3	t_4
`-   return (a <= b) ? b : (Float.isNaN(a + b) ? Float.NaN : b);`	●	●	●	●
`+   return (a <= b) ? b : (Float.isNaN(a + b) ? Float.NaN : a);`				
Qualitative State: `Float.isNaN(_return)`	●			
Qualitative State: `!Float.isNaN(_return)`		●	●	●
Qualitative State: `_return >= a && _return >= b`		●	●	
Qualitative State: `_return < a || _return < b`				●
`  }`				
Test outcome (e):	✓	✓	✓	✗

Figure 4.11: Math 59 patch snippet with manual landmarks.

straightforward way—automatically derive a partitioning scheme able to isolate faults, but that can be added by developers without much effort—in the form of *pre-* and *post-conditions*, for instance.

Figure 4.10 depicts a method from the Closure compiler responsible for an optimization that removes *unreferenced* arguments from a function definition. Method `removeUnreferencedFunctionArgs` contains an omission error—it must check the field variable `this.removeGlobals` before performing the optimization. This method, along with others that depend on the value of `this.removeGlobals`, is an

		t_1	t_2	t_3	t_4
	`public Paint getPaint(double value) {`				
	Qualitative State: `value > this.upperBound`				●
	Qualitative State: `value < this.lowerBound`			●	
	`double v = Math.max(value, this.lowerBound);`	●	●	●	●
	`v = Math.min(v, this.upperBound);`	●	●	●	●
-	`int g = (int) ((value-this.lowerBound)/(this.upperBound-this.lowerBound)*255);`	●	●	●	●
+	`int g = (int) ((v-this.lowerBound)/(this.upperBound-this.lowerBound)*255);`				
	`return new Color(g, g, g);`	●	●	●	●
	`}`				
	Test outcome (e):	✓	✓	✗	✗

Figure 4.12: Chart 24 patch snippet with manual landmarks.

appropriate target for a *pre-condition* check—which, as mentioned above, can be encoded as a set of qualitative *SFL* components.

Figure 4.11 shows a simple method to calculate the maximum value of two `float`-type values. Besides a simple value comparison, additional logic is required for the case when at least one of the arguments is *not-a-number* (`NaN`). This method is buggy since it returns `b` even when its value is less than `a`. It fails, therefore, to ensure the *post-condition* that the returned value is greater than or equal to both method arguments.

Lastly, in Figure 4.12, the manual landmark added to the faulty method's parameter checks if its value lies within previously defined upper and lower bound fields. In this case, a *conjunction* of the qualitative states `value > this.upperBound` and `value < this.lowerBound` correlates with test failures.

4.3.3 Java Implementation of Landmarking Framework

We have implemented a framework for landmarking and instrumenting Java code to facilitate the collection of *QR*-augmented spectra[1]. The inner workings of the framework—depicted in Figure 4.13—are detailed below.

[1]Available at `https://github.com/aperez/q-sfl`.

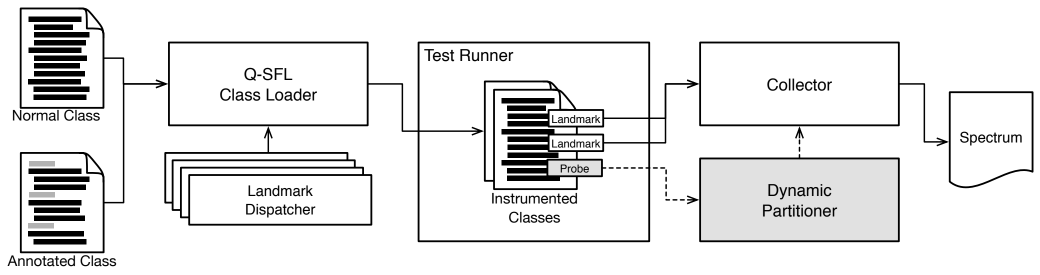


Figure 4.13: Java-based qualitative landmark collection framework.

Listing 1 Example of source code with landmarking annotations.

```
@LandmarkWidth(DefaultDispatcher.class)
public class Calculator {
  public static int add(@IntegerHandler int a,
      @Skip int b) {
    return a + b;
  }

  public static int sub(int a,
      @Landmark(handler=CustomHandler.class) int b) {
    return a - b;
  }
}
```

The framework allows for the creation, instrumentation and collection of both manual and automated landmarks of method parameters and return values. With manual landmarking, users can create specific *landmark handlers* for data objects and manually annotate their source code to instruct the framework what should be instrumented. The framework also provides generic, static landmark handlers, as well as data probes that enable dynamic landmarking.

Our framework relies on Java *annotations*—a way to syntactically bind metadata to source code—to specify *what* variables to partition, and *how*. At runtime, the annotations are read by the framework during *class-loading*, prompting it to instrument the necessary landmark collection code into the target class' bytecode. Because of the fact that we use *annotations* to declare how to create landmarks, and because the code injection happens at runtime, our instrumentation can be turned off at any time without the need to modify the code or recompile it.

Listing 1 shows a small Java code snippet exemplifying the use of our framework's landmarking annotations. The class-level annotation, `@LandmarkWidth`, is used to specify a landmark handler *dispatcher*, responsible for assigning landmark handlers to all method parameter types within the target class. The `DefaultDispatcher` is a *dispatcher* that invokes the aforementioned default automated handlers to partition the domain of primitive types and objects using simple sign-partitioning or *null*-checking strategies. Users are free to write their own custom, domain-specific *dispatchers*.

Besides *dispatchers*, users can assign *handlers* to specific parameters, as is the case of parameter `a` from method `add`, using `@IntegerHandler`—the generic automated handler for integer values. Every handler annotation specified this way takes precedence over the mapping specified in a *dispatcher*. A given parameter can also be skipped with a `@Skip` handler. Lastly, custom, manual landmark handlers are inserted with the `@Landmark` annotation, pointing to the class responsible for implementing the custom handler. Note that instrumentation is also possible without the use of

Listing 2 Implementation of a custom handler for landmarking integer values.

```java
public class CustomHandler implements Handler {
  public int states() {
    return 3;
  }

  public int handle(Object o) {
    Integer i = (Integer)o;
    if (i == 0) { return 0; }
    return i > 0 ? 1 : 2;
  }

  public String landmarkName(int l) {
    if (l == 0) { return "zero" };
    else if (l == 1) { return "positive" };
    return "negative";
  }
}
```

annotations. In classes without any landmarking annotations, the framework relies on the landmark dispatcher configured as the default for such cases.

Listing 2 shows the user-level implementation of a landmark handler. Its `states` method returns the number of qualitative states resulting from the partitioning; the `handle` method is invoked with the value to be inspected, returning the respective qualitative state identifier; and the optional `landmarkName` method is intended to provide users with more contextual information about each qualitative state identifier.

While landmark handlers are used to implement manual and static landmarking strategies, our framework also provides the ability to specify *data probes* that will store the runtime value of its target variable per method invocation, and per transaction. This is to enable the use of clustering techniques for dynamic landmarking. We should note that the data partitioning step is currently done outside of the framework context; and that in order to gather data probes, users must ensure that their target variables can be serialized.

Lastly, all landmarks are gathered by the framework's collector module, responsible for creating a spectrum representation that *SFL* tools—such as GZOLTAR [Campos et al., 2012]—can understand and diagnose.

4.4 Evaluation

To evaluate our approach, we compare the cost of diagnosing a collection of faulty software programs using regular spectra against using *QR*-augmented spectra. We aim to answer the following research questions:

Research Question 4.1

Does augmenting spectra with qualitative components improve their diagnosability?

Research Question 4.2

Is there a particular automated landmarking strategy that consistently shows improved diagnosability?

In **RQ4.1** we are concerned with finding out if *there exist* qualitative partitionings able to improve the fault localization ranking to the extent that faulty components are inspected earlier—thus decreasing *wasted effort* in a debugging task. If **RQ4.1** is true, it prompts us to **RQ4.2**, which asks whether there is an automated partitioning scheme able to qualitatively enhance spectra in an automated way—without the need of user interaction or domain knowledge.

4.4.1 Methodology & Evaluation Metric

Our methodology for this empirical evaluation is as follows. We run the fault-revealing test suite of each DEFECTS4J subject, gathering per-test branch-level coverage and test outcomes, to be able to generate spectra. Besides coverage, we also record *primitive-type* parameter and return values for every method call in each test execution. This enables us to experiment with different qualitative partitioning strategies in an *offline* manner.

Using the recorded argument and return value data, we create multiple (automated) partitioning models resulting in several *Q-SFL* variants. A *static partitioning* variant using automated sign partitioning based on the variable's type, as described in Section 4.3.1, was considered. For *dynamic partitioning*, several clustering and classification algorithms[2] were considered:

Sign resorts to sign-partitioning, as described in Section 4.3.3.

X-means clustering algorithm, a variation of k-means, that iteratively increases the number of clusters until a model selection criterion—such as the Bayesian information criterion—is reached [Pelleg and Moore, 2000].

k-NN classification algorithm, where objects are assigned to the class that is most common among their k nearest neighbors.

Linear classification, assignments are based on the value of a linear combination of the features.

[2]We chose popular classification algorithms [Han et al., 2011] implemented in the `Scikit-learn` package. X-means, as implemented in the `pyclustering` package, was selected as it can automatically decide the optimal number of clusters to use [Pelleg and Moore, 2000].

Logistic regression, predicts the probability that an observation falls into one of two categories.

Decision Tree classifier, learns tree-like model of decisions, where leaf nodes represent class labels.

Random Forest classifier, constructs a group of decision trees and outputs the mode of the individual trees' outputs.

Using the recorded runtime data, we construct individual models for each *primitive-type* method parameter and each method *primitive-type* return value in a DEFECTS4J subject, leveraging every aforementioned strategy. Test outcomes are used as the class labels in the case of supervised models. It is then possible to create a *QR*-augmented spectrum per strategy, by feeding the recorded per-test value observations to the respective models and checking which class they fall into (thus *emulating* the assignment of a qualitative state). Note that we are not using models for *prediction*, but rather as a partitioning scheme based on observed values, therefore we do not break our data into *training* and *test* sets, as is customary in *prediction* scenarios. Because we use automated, domain independent partitioning, only primitive types are considered in the evaluation.

For each DEFECTS4J subject, we choose as the landmarking strategy to consider in the evaluation the one that is able to create the largest set of distinct, non-ambiguous qualitative components out of the faulty method(s). All spectra are diagnosed using reasoning-based *SFL*, following the methodology described in Section 1.2.2. All tools and scripts used in this experiment, as well as gathered data, are available at `https://github.com/aperez/q-sfl-experiments`.

To be able to effectively compare a *QR*-augmented spectrum against its respective original spectrum (*i.e.,* the one without components stemming from qualitative landmarking), we first reduce the diagnostic report to *method* components. This reduction is done by considering the highest fault probability of any subcomponent belonging to each method, to effectively be able to compare method-level diagnostic effort between the two approaches. A change in diagnostic effort is measured using

$$\Delta C_d = C_d(\text{Original}) - C_d(QR\text{-Augmented}) \tag{4.1}$$

where C_d is the cost of diagnosis. A positive ΔC_d means that the faulty component has risen in the ranking reported by *SFL* techniques when *QR* is used, yielding a lowered cost of diagnosing. A ΔC_d equal to zero means that the effort to diagnose the faulty component is the same in the two approaches. Negative values of ΔC_d mean that there is an increased effort to pinpoint the fault when compared to the original spectrum.

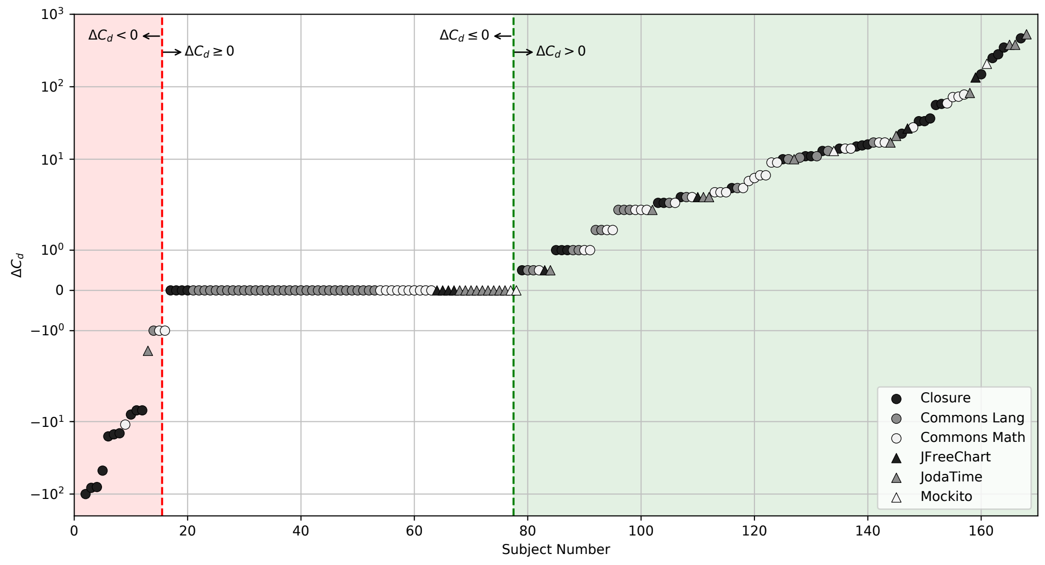


Figure 4.14: Difference in C_d between original and QR-augmented spectra per subject.

4.4.2 Results

We were able to automatically partition the faulty method in 167 DEFECTS4J subjects. The remaining DEFECTS4J subjects were discarded because (i) the faulty method does not contain parameters nor does it return a value; because (ii) the faulty method only contains non-primitive, non-null, complex-typed parameters, which cannot be handled by the set of partitioning strategies described in Section 4.4.1; or because (iii) the aforementioned partitioning strategies were unable to create qualitative states whose coverage differs from their enclosing method[3].

The breakdown of selected partitioning strategies per subject is as follows: Sign: 102 (61%); X-means: 25 (15%); k-NN: 8 (5%); Linear Regression: 1 (1%); Logistic Regression: 4 (2%); Decision Tree: 11 (7%); Random Forest: 16 (10%). Our sign-partitioning default strategy was used to qualitatively enhance the majority of considered subjects, while other strategies such as linear classification and logistic regression were rarely selected. We suspect the reason that supervised learning approaches—which were fed test case outcomes as the target class label—only exhibited superior performance in 40 subjects (24%) is due to the fact that the number of failing tests in test suites is often much smaller than the amount of passing tests, weakening the learned partitioning model.

Figure 4.14 shows a scatter plot with the ΔC_d of all subjects under analysis. Shown in a red background are the 15 subjects (9%) with a negative ΔC_d—meaning that the report has suffered a decrease in accuracy after augmenting the spectra. The majority of these subjects belong to the Closure project. The 62 subjects (37%) with

[3]That is, subjects where all qualitative components either are never active or have exactly the same activation pattern as the enclosing component are discarded since these subjects will invariably produce a ΔC_d of 0.

Table 4.1: Statistical tests.

	Original Spectra	***QR*-augmented Spectra**
Mean C_d	60.28	37.56
Median C_d	6.00	2.50
C_d Variance	2.10×10^4	1.56×10^4
Shapiro-Wilk	$W = 0.46$ p-value $= 2.20\times10^{-22}$	$W = 0.32$ p-value $= 1.10\times10^{-24}$
Wilcoxon Signed-rank	$Z = 5.45$ p-value $= 5.10\times10^{-10}$	

$\Delta C_d = 0$, where the faulty component has remained in the same position of the ranking, are shown in a white background. Lastly, 90 subjects (54%) that exhibited a positive ΔC_d—cases where *QR*-augmented spectra improved diagnosability—are shown in green. All in all, *Q-SFL* is at least as good as the original approach in 92% of scenarios.

The augmented spectra of such subjects where $\Delta C_d < 0$ contains *non-faulty* qualitative states (in the sense that they do not belong to the faulty method) that have a high correlation to failing test outcomes, which end up rising in the diagnostic ranking. Conversely, subjects that exhibit $\Delta C_d > 0$ contain qualitative components are able to partition the faulty method's coverage in a way that is highly correlated with failing tests, not only lowering the effort to diagnose, but providing some more context for the fault by way of their qualitative descriptions. When $\Delta C_d = 0$, the qualitative partitioning strategy did not manage to expose any qualitative state that is highly correlated with the *error vector*.

Table 4.1 presents statistics computed to assess whether the observed metrics yield statistically significant results. The first two rows show the mean and median C_d values for both spectra in our analysis. As to be expected, *QR*-augmented spectra exhibits an overall lower effort to diagnose when compared to the original spectra. *QR*-augmented spectra also have less variance. To assess if the findings are *statistically significant*, we first performed the Shapiro-Wilk test for normality of effort data [Shapiro and Wilk, 1965] in both the original spectra case and *QR* case. With 99% confidence, the test's results—which can be seen in the fourth row of Table 4.1—tell us that the distributions are not normal. Given that C_d is not normally distributed and that each observation is paired—for each subject, there is an original spectrum and a QR-augmented spectrum—, we use the non-parametrical statistical hypothesis test Wilcoxon signed-rank [Wilcoxon, 1945]. Our null-hypothesis is that the median difference between the two observations (*i.e.*, ΔC_d) is zero. The fifth row from Table 4.1 shows the resulting Z statistic and

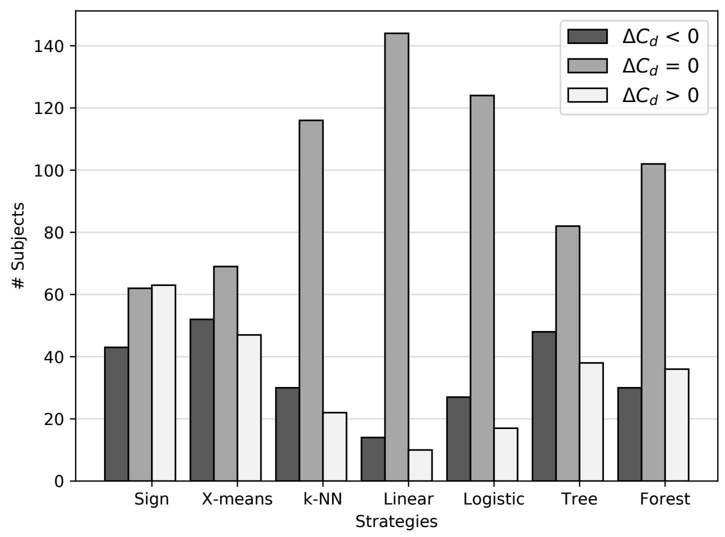


Figure 4.15: Breakdown of diagnostic performance per partitioning strategy.

p-value of Wilcoxon's test. With 99% confidence, we refute the null-hypothesis. Revisiting **RQ4.1**:

Research Question 4.1

Does augmenting spectra with qualitative components improve their diagnosability?

Answer: Yes, augmenting faulty spectra with new components resulting from qualitative landmarking of method parameter (and method return) values yields a *statistically significant* improved diagnostic report.

To be able to answer **RQ4.1**, we have selected for each subject the strategy with the highest number of qualitative partitions targeting the faulty method, as we were only concerned with the *existence* of *a* partitioning strategy that would improve diagnosability. However, in practice, it is not realistic to know *a-priori* what the faulty method is[4]. Figure 4.15 shows a breakdown of the number of subjects that fall into the $\Delta C_d < 0$, $\Delta C_d = 0$ and $\Delta C_d > 0$ categories for every partitioning strategy considered in this evaluation. The bar plot tells us that no single strategy achieves the same number of positive ΔC_d scenarios as the partition cardinality selection criterion employed to answer **RQ4.1** and to produce Figure 4.14. Furthermore, strategies that were often picked by that criterion (namely, the default and X-means strategies) also show an increased number of ΔC_d scenarios when compared to others. This leads us to conclude that no single strategy (out of the ones that were analyzed) is able to *consistently* show improved diagnoses. Revisiting **RQ4.2**:

[4]Although some effort has been put forth to hierarchically debug programs using *SFL* [Perez, 2012; Perez et al., 2014].

Research Question 4.2

Is there a particular automated landmarking strategy that consistently shows improved diagnosability?

Answer: No, at least for the automated landmarking strategies considered in the evaluation, there is no evidence that a single automated strategy (which only partitions primitive-typed data, as per this evaluation) can consistently outperform the original spectra. However, since *Q-SFL* can improve diagnosability, as per the answer to **RQ4.1**, we presume that *manual* or more complex, context-aware, automated *white-box* strategies—which can perform static and dynamic source code analysis on custom data types, expanding the scope of landmarking strategies—are more suited to outperform the original spectra due to more *effective* and more *informed* partitioning.

4.4.3 Threats to Validity

We now outline the potential threats to the validity of the experiment detailed above:

Construct validity In Section 4.4.1, we justify the use of automated black-box partitioning techniques stemming from machine learning applications by stating that their resulting models emulate users' manual landmarking strategies. In reality, this may only be a rather crude and simplistic approximation of user behavior, as black-box techniques are only fed with runtime information (such as concrete parameter values and test outcomes). However, we argue that manual landmarking—and automated white-box landmarking alike—may be able to achieve a better, more informed qualitative partitioning due to the fact that the source code can be analyzed.

Internal validity There is a possibility that, due to the complexity of our spectrum-gathering and landmarking instrumentation framework, there may remain an implementation bug somewhere in the codebase. To mitigate this, we extensively reuse thoroughly-tested code from the Java-based spectrum-gathering tool GZOLTAR.

External validity A potential threat to the external validity relates to the fact that the chosen set of subjects may not be an accurate representation of all bugs that can happen during development or of different software development methodologies. We attempt to reduce the selection bias by leveraging an established collection of real, reproducible faults that have occurred in six medium to large-sized open source software applications (the DEFECTS4J dataset), which is extensively used in the evaluation of related work [Pearson et al., 2017; Campos et al., 2013; Shamshiri et al., 2015]. Understandably, this public dataset does not contain any closed-source codebases whose development practices may differ from open source software. We also aim to ensure the reproducibility of our evaluation by providing the source of our instrumentation tool, the scripts used to run the evaluation, and all data gathered.

4.5 Discussion

In this section we discuss our findings from the empirical evaluation, as well as outline their practical implications:

- Our evaluation has revealed that augmenting program spectra through qualitatively partitioning variables by their value does increase the effectiveness of *SFL* reports. Figure 4.14 also shows that qualitative reasoning has the potential to substantially decrease the cost of diagnosis, allowing the fault to rise in the ranking by as much as 10^3 components in real-world, open source, buggy projects.
- The criterion for choosing the best partitioning strategy involved assessing the number of partitions of the faulty method. This is not applicable in practice, as the faulty method is not known. Without such selection criterion, the use of a single black-box partitioning strategy yields either an increased number of negative ΔC_d cases, or a decreased number of ΔC_d cases.
- Although we mention when answering **RQ4.2** that we see no evidence of a black-box partitioning strategy that is *consistently* more accurate at pinpointing faults than the original spectrum, there may still exist a partitioning scheme—perhaps an ensemble of partitioning strategies—able to outperform the original spectrum. We plan to investigate this hypothesis as future work.
- By showing that diagnostic improvements are possible through *QR*, this work paves the way for future efforts, namely by leveraging manual and automated white-box landmarking instead of the (simplistic) black-box methods explored in the evaluation. In particular, we believe that using techniques such as abstract interpretation [P. Cousot and R. Cousot, 1977] and other static/dynamic analyses—to perform a more thorough boundary value analysis and equivalence-class partitioning—will likely produce better diagnostic results, both in terms of accuracy as well as comprehension.

4.6 Related Work

There have been previous forays into enhancing the diagnostic report of automated fault localization techniques to either improve their accuracy or comprehension of the failing component. This section outlines some of the efforts in these research areas.

SFL approaches to debugging typically present their report to users as a list of suspicious components that is sorted according to the likelihood of being faulty. Jones et al. [2002] and Jones and Harrold [2005] proposed a *visual* way of depicting the results of a similarity-based *SFL* diagnosis. The TARANTULA tool presents an interactive visualization that shows the entire codebase under analysis, and color-codes each code fragment according to their *SFL* suspiciousness score, ranging from red (high suspiciousness) to green (low suspiciousness). Campos et al. [2012] presented the GZOLTAR tool, which expands on the visual concept by leveraging tree-based visualizations that innately exploit the tree-like structure of Java code, naturally aggregating neighboring components and aiding exploration of suspicious code regions. In a user study by Gouveia et al. [2013], the authors showed that visual encoding of *SFL* diagnostic reports does improve their effectiveness, as it provides users with more structural context to perform the debugging task.

Another approach to improve the comprehension of faults was proposed by Ko and Myers [2008]. The approach, called Whyline, allowed the users to obtain evidence about the program's execution before forming an explanation of the cause by providing the ability to ask "why did" and "why didn't" questions about program output. Users were able to explore answers to such questions using a combined timeline visualization, bookmarking tool and navigational aid. The authors performed a user study with a prototype able to analyze Java projects, and showed that Whyline users managed to diagnose faults three times as often and were two times as fast compared to the control group [Ko and Myers, 2009].

De Souza and Chaim [2013] proposed an extension to *SFL* to improve comprehension. It uses *integration coverage* data—by way of capturing method invocation pairs—to guide the fault localization process. By calculating the fault suspiciousness of component pairs, the authors were able to generate *roadmaps* for code investigation, guiding users through likely faulty paths and increasing the amount of contextual cues.

Advancements in bug prediction [D'Ambros et al., 2010; Bettenburg et al., 2012; F. Zhang et al., 2016]—*i.e.*, the creation of predictive models that assess whether a software artifact is likely to contain bugs by relying in diverse information, such as *code metrics* [Nagappan et al., 2006], *process metrics* [Moser et al., 2008], *previous defects* [Kim et al., 2007], *etc.*—enabled its use within automated fault localization processes. S. Wang and Lo [2014] proposed an ensemble approach to fault localization that exploits information from versioning systems, bug tracking repositories and structured information retrieval from the source code. Cardoso and Abreu [2013a] relied on kernel density estimation models of component behavior and previous diagnoses to better estimate the component goodness parameter in spectrum-based reasoning. Elmishali et al. [2016] also modified the traditional spectrum-based reasoning framework by leveraging a fault prediction model trained with historical information from the project's versioning system and bug tracker to compute the *prior probability distribution* of diagnostic candidates. Sohn and Yoo [2017] used similarity-based *SFL* suspiciousness values, as well as source code metrics—code size, age and code churn—as features for two learn-to-rank techniques: genetic programming and linear rank support vector machines.

Augmenting fault-localization via slicing has also been proposed. Mao et al. [2014] proposed the use of *dynamic backward slices*—comprised of statements that directly or indirectly effect the computation of the output value through data- or control-dependency chains [Korel and Laski, 1988]—as components in similarity-based *SFL*. Hofer and Wotawa [2012] propose an approach that leverages a model-based slicing-hitting-set-computation [Wotawa, 2010]—which computes the dynamic slices of all faulty variables in all failed test cases, derives minimal diagnostic candidates from the slices and computes fault probabilities for each statement based on number of the diagnoses that contain it. Similarity-based *SFL* scores are used as *a-priori* fault probabilities for the model-based technique.

4.7 Summary

In this chapter, we propose a methodology to mitigate the limitations concerning the abstract nature of program spectra, which are outline in Section 1.3.3. Concretely, in this chapter:

- We proposed the *Q-SFL* approach (Section 4.3, page 70):
  - It leverages concepts from *QR* (Section 4.2, page 69), which models complex systems by describing continuous values by their discrete, behavioral qualities, so that we can simulate and reason about a system without exact quantitative information;
  - Our approach inspects the runtime behavior of parameters and return values, and partitions their domains into qualitative states. Each qualitative state is subsequently treated as an *SFL* component, thereby providing the *SFL* framework with more qualitative information and also increasing fault isolation.
- We demonstrated several examples of how different domain partitioning strategies can be used to improve the accuracy of traditional *SFL* (Section 4.3.2, page 72).
- We conducted an empirical evaluation comparing the accuracy of *Q-SFL* versus traditional *SFL* on real faults from the DEFECTS4J catalog (Section 4.4, page 79). Results show that:
  1. Spectra which were *augmented* using qualitative partitioning of method parameters show a (statistically significant) improvement in the diagnostic accuracy (answering **RQ4.1**, page 84);

2. However, out of the partitioning strategies we considered, we have found none that performed consistently better than traditional *SFL* (answering **RQ4.2**, page 85). This result leads us to conclude that more intricate partitioning strategies (such as *ensembles* of strategies, or *white-box* techniques to partition complex types) will be required to effectively improve diagnostic accuracy.

Program Comprehension as a Diagnostic Problem

5

A Diagnosis-based Approach to Software Comprehension
Alexandre Perez and Rui Abreu
In: Proceedings of the 22nd International Conference on Program Comprehension, ICPC 2014, Hyderabad, India, June 2-3, pp. 37–47, 2014.

Framing Program Comprehension as Fault Localization
Alexandre Perez and Rui Abreu
In: *Journal of Software: Evolution and Process* 28.10, pp. 840–862, 2016.

Abstract *Program comprehension is a time-consuming task performed during the process of reusing, re-engineering, and enhancing existing systems. Currently, there are tools to assist in program comprehension by means of dynamic analysis. However, most cannot identify the topology and the interactions of a certain functionality in need of change, especially when used in large, real-world software applications. We propose an approach, coined Spectrum-based Feature Comprehension (SFC), that borrows techniques used for automatic software fault localization. SFC analyses the program by exploiting run-time information from test case executions to identify the components responsible for implementing a given targeted feature, helping software engineers to understand how a program is structured. We present a toolset, coined* PANGOLIN*, that implements SFC and displays its report to the user using an intuitive visualization. A user study with the open-source application Rhino is presented, demonstrating the efficiency of* PANGOLIN *in locating the components that should be inspected when changing a certain functionality. Participants using SFC spent a median of 50 minutes locating the feature with greater accuracy, whereas participants using coverage tools took 60 minutes. Finally, we also detail the Participatory Feature Detection (PFD) approach, an extension of SFC, where user interactions with the system are captured, removing the hinderance of requiring pre-existing automated tests.*

5.1 Introduction

Software maintenance is a crucial part of software engineering. The need to add or change features in existing software applications is becoming more and more prevalent. Furthermore, the ever increasing complexity of software systems and applications renders software maintenance even more challenging. One of the most daunting tasks of software maintenance is to understand the application at hand [Corbi, 1989]. In fact, recent studies point out that developers spend 60%

to 80% of their time in comprehension tasks [Tiarks, 2011]. During this program understanding task, software engineers try to find a way to make both the source-code and the overall program functionality more intelligible. One of these ways is to create a *"mental map"* of the system structure, its functionality, and the relationships and dependencies between software components [Lange and Nakamura, 1997; Renieris and Reiss, 1999].

To fully understand how a software application behaves, software engineers need to thoroughly study the source-code, its documentation and any other available artifacts. Only then do engineers gain sufficient understanding of the application, enabling them to seek, gather, and make use of available information to efficiently conduct maintenance or evolution tasks. This *program comprehension* (also known as *program understanding/software comprehension*) phase is thus resource and time consuming. In fact, studies show that up to 50% of the time needed to complete maintenance tasks is spent on understanding the software application and gaining sufficient knowledge to change the desired functionality [Corbi, 1989]. Currently, there are several approaches that focus on dynamic analysis to provide visualizations of the software system, identifying their components and their relationships, *e.g.*, [Pauw et al., 1998; Reiss and Renieris, 2001; Greevy et al., 2006]. However, these approaches may not clearly show what code regions the developer needs to inspect in order to change a certain functionality. Another problem regarding these dynamic analyses is the fact that program traces of sizable programs encompass large amounts of data [Zaidman, 2006].

We propose an approach, coined **Spectrum-based Feature Comprehension (*SFC*)**, that departs from related work by leveraging techniques from the software fault localization domain (namely *SFL*), which were shown to be efficient, even for large, resource-constrained environments [Abreu et al., 2009d]. Fault localization tasks therefore exhibit many similarities with maintenance or evolution tasks—which involve pinpointing features among the code in order to change a particular application behavior. Thus, our *SFC* approach maps *SFL* to the problem of feature localization to provide efficient program comprehension and dependency visualization, thereby decreasing developer effort.

To assess the effectiveness of our approach, we have conducted a user study with the open-source project Rhino. It demonstrates the accuracy and effectiveness of the *SFC* approach in aiding users to pinpoint the components that need to be inspected when evolving/changing a certain feature in the application. It also shows that users spend less time locating features with *SFC* when compared to using standard code coverage tools. In the case of the user study, participants using *SFC* spent a median of 50 minutes locating a feature, whereas participants using coverage tools took 60 minutes.

We also address a potential drawback of the *SFC* approach by extending it to allow **Participatory Feature Detection (*PFD*)**. To pinpoint a feature with *PFD*, developers are asked to interact with the application and to record whether the feature they want to locate was involved or not in each execution. This way, *SFC* becomes applicable to any interactive application, without requiring a test suite to obtain the execution traces required by spectrum-based techniques.

The chapter makes the following contributions:

- We describe *SFC*, an approach that, similar to fault-localization techniques, exploits run-time information from system executions to identify dependencies between components, helping software engineers in understanding how a program is structured.
- We provide a toolset, PANGOLIN, which generates a visualization of associated and dissociated components of an application functionality.
- A user study with a large, real-world, software project, demonstrating the effectiveness of our approach in locating the components that should be inspected when evolving/changing a certain feature.
- We extend the approach to allow participatory feature detection: users can manually collect and label executions on any interactive project, removing the hinderance of requiring pre-existing automated tests.
- We propose an accuracy metric to evaluate the effectiveness of the *SFC* approach.

5.2 Spectrum-based Feature Comprehension (*SFC*)

The *SFC* approach to be presented in this section is in essence a mapping between feature detection and comprehension concepts and spectrum-based fault localization concepts, enabling the use of *SFL* in software maintenance or evolution scenarios.

5.2.1 Mapping Feature Detection to Fault Localization

As argued in Section 1.3.4, locating *features*—*i.e.*, portions of source-code responsible for implementing a given functionality (Definition 10)—closely resembles the act of locating the root cause of system failures, as the objective of both tasks is to pinpoint in the code the origin of a particular *behavior*. To leverage fault localization techniques in the feature detection context, we first present a mapping between the two domains.

When trying to evolve/modify a certain feature f, one is interested in the runtime relationships and interactions between f and other system components. As such, one should not use the error vector e—which is responsible for cataloging erroneous behavior—as with traditional *SFL* approaches presented in Section 1.2. Instead, we propose the use of an *evolution vector* ev_f when pinpointing features:

Definition 11 (Evolution Vector) *The evolution vector ev_f is an N-length binary vector. In this vector, a given position j is true (*i.e.*, $ev_f(j) = 1$), if the j^{th} transaction executes feature f.*

While the error vector captures whether transactions have passed or failed, the evolution vector ev_f captures whether feature f was involved or not in each transaction. Methodologies for actually gathering ev_f for a given feature detection task will be detailed in Sections 5.3 and 5.4. The resemblance between the evolution vector and system components is captured by the association measure:

Definition 12 (Association Measure) *The association measure $a_f(i)$ indicates the degree of correlation between a component i's activity (*i.e.*, $\{\mathcal{A}_{ij} | j \in 1..N\}$) and the evolution vector ev_f.*

Association measures for each component i therefore correspond to the outcome of the *SFL* technique (*i.e.*, similarity scores or fault probabilities, depending whether similarity-based or reasoning-based techniques are used) given an activity matrix $\mathcal{A}$ and the evolution vector ev_f as input. We assume that association measures range between 0 and 1, meaning that in the case of similarity-based *SFL*, one should normalize the fault predictor score if necessary. For conciseness, we will be using similarity-based *SFL* with the Ochiai predictor throughout this chapter.

When evolving a feature f, it is important to inspect its associated components because they may either call f or be called by f, and thus may need to be modified in accordance with the changes made to f.

Definition 13 (Associated Component) *A component i is* associated *with f if its association measure $a_f(i)$ is close to 1. This means that when f is executed, i is likely to be involved.*

In contrast to associated components, if a component is dissociated to f, it does not need to be inspected when f is modified.

Definition 14 (Dissociated Component) *A component i is* dissociated *to f if its association measure $a_f(i)$ is close to 0. This means that when f is executed, i is not likely to be involved.*

Components with association measures of neither 0 nor 1 should be inspected, but only modified with great care. This is because these components are shared among features. We will revisit this assertion while evaluating the approach in Section 5.3.

5.2.2 PANGOLIN Toolset

We now introduce the PANGOLIN toolset[1], an Eclipse plugin that implements the *SFC* approach and displays its results with the aid of a sunburst visualization. Its an extension of the GZOLTAR toolset which enables feature localization.

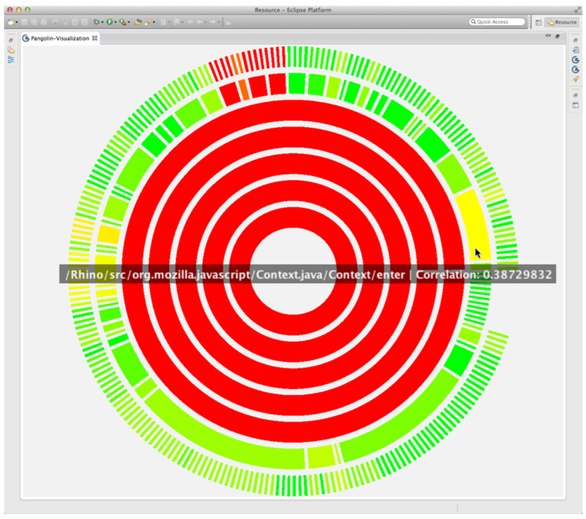

Resource – Eclipse Platform
Pangolin-Visualization
/Rhino/src/org.mozilla.javascript/Context.java/Context/enter | Correlation: 0.38729832

Figure 5.1: PANGOLIN's sunburst visualization.

The PANGOLIN plugin performs a very lightweight dynamic analysis by instrumenting the project, so that the activity matrix is gathered during runtime. The plugin uses the project's JUnit test cases as the set of system runs. In order to perform the analysis, users are prompted to identify which system runs exercise the feature under consideration. Subsequently, PANGOLIN plugin computes *SFC*'s feature-association measure for every component in the project, and displays that information in a sunburst visualization, as shown in Figure 5.1. This sunburst visualization depicts the current project's topology in a hierarchic fashion, starting from the root component representing the whole project in the inner circle, up to individual lines of code in the outer circle. Each component is color coded with the corresponding association measure, ranging from bright green if the association measure is close to zero; to yellow if the association measure is close to 0.5 and to red if it is close to 1. When a user hovers the mouse on a component, a label identifying that component and its association measure is shown, as depicted in Figure 5.1. By selecting a particular component, Eclipse's code editor will open and the cursor is positioned on the start of the chosen component. Sunburst was shown to be effective at conveying

[1]PANGOLIN is available online at `www.gzoltar.com/pangolin`.

actionable diagnostic information (more so than mere list-based reports) in a user study by Gouveia et al. [2013].

Intuitiveness of the Sunburst Visualization Lawrance et al. [2013] have applied information-foraging theory to explain the intuitiveness of such visualization approaches and their effectiveness in conveying diagnostic reports. Information-foraging theory aims to both *"explain and predict how people will best shape themselves for their information environments and how information environments can be shaped for people"*, as defined by Pirolli [2007]. This theory is itself based on optimal foraging theory, which tries to explain the behavior of *predators* and *preys*. *Predators* try to find *preys* by following their *scent*, and *preys* are more likely to be in places (or *patches*) where the *scent* is more intense. In the information-foraging context, the *predators* are the people in need of information and the *preys* are the information itself. The *scent* is the interpretation of the environment by the *predators*. In the context of feature detection, we consider the following mapping:

- *Predator* is the person performing the maintenance task;
- *Prey* is what the *predator* seeks to know to pinpoint the code regions that need to be changed;
- *Proximal cues* are the runtime behaviors that suggest scent related to the *prey*;
- *Information scent* is the *predator* interpretation of the *SFC* report;
- *Topology* is the collection of navigable links between in the visualization.

Information-foraging theory assumes that the developer's choices are an attempt to maximize the information gain per navigation interaction's cost [Fleming et al., 2013]. Since these are not known by developers *a priori*, their decisions will be based on the *expected* gain and cost. Perez and Abreu [2013] have shown that the sunburst visualization, along with its interaction features, can indeed reduce the cost of navigating through the various system components (be it packages, classes, methods, even statements). By color coding each system component, proximal cues are also provided, guiding the developer towards likely associated regions of the source-code. At the same time, regions that should not be explored (where, *e.g.*, relevant executions were not active) are also distinctively highlighted. Hence, a better information scent is conveyed to the developer, increasing the information gain.

5.3 User Study

In this section, we evaluate the effectiveness of *SFC* when applied to a real-world application—the Rhino project. This user study aims to answer the following research questions:

Research Question 5.1

Can programmers using PANGOLIN pinpoint a feature more accurately than by inspecting standard test coverage traces?

Research Question 5.2

Is the time taken to pinpoint a feature with PANGOLIN at least comparable to inspecting standard test coverage traces?

In **RQ5.1**, we are concerned about assessing the effectiveness of the technique and our tool. **RQ5.2** tries to ensure that our approach will not negatively impact existing program comprehension processes.

First, we describe the subject of our evaluation and the setup for our user study. Afterward, we present the results of the user study and potential threats to validity.

5.3.1 The Rhino Project

The software application under consideration for this case study is the open-source project Rhino[2]. Rhino is a Javascript engine written entirely in Java, maintained by the Mozilla Foundation. It is typically embedded into Java applications to provide scripting to end users. It also allows Javascript programs to leverage Java platform APIs. Rhino automatically handles the conversion of Javascript primitives to Java primitives, and vice versa (*e.g.*, Javascript scripts can set and query Java properties and invoke Java methods). Rhino is comprised by 28 packages, 433 classes and 75170 source lines of code. Furthermore, this project contains 441 unit tests, written for the JUnit framework, which cover 56% of the project's statements and 45% of branches (9,323 out of a total 20,802 branches).

5.3.2 User Study Setup

The user study was performed by 108 students enrolled in the Software Engineering course of the Master in Informatics and Computing Engineering program from the Faculty of Engineering of University of Porto. The experiment was performed in the context of a lab session for the Software Engineering course. A pre-requisite for enrollment in that course is that students must have completed the following courses: 'Programming Fundamentals', 'Algorithms and Data Structures' and 'Object-Oriented Programming Lab', which means that all participants had at least three years of experience with the Java programming language and were familiar with both the Eclipse IDE and the JUnit testing framework. Additionally, no participants had used Rhino before. Participants were grouped into pairs to perform the requested

[2]Available at `https://developer.mozilla.org/en-US/docs/Rhino`

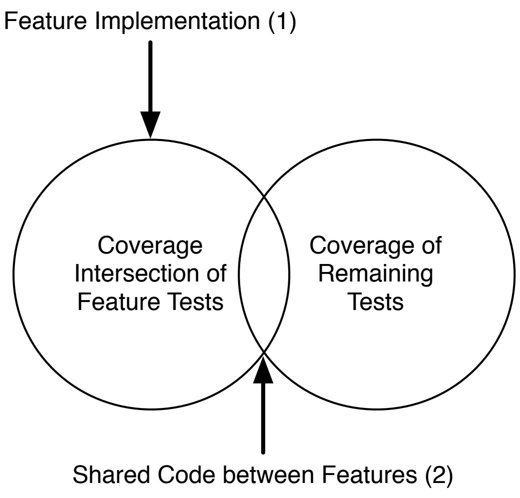

Feature Implementation (1)
Coverage Intersection of Feature Tests
Coverage of Remaining Tests
Shared Code between Features (2)

Figure 5.2: Feature analysis with a coverage tool.

task, which was also a course requirement. Each pair had access to one computer to perform the task. They were not compensated for performing the experiment. However, with the experiment being performed in a lab session, its attendance was mandatory.

The requested task was the following. Participants were requested to identify source-code regions that exclusively implement a certain feature (labeled as task (T1)), and also regions where that feature is being used (*i.e.*, code regions shared among different features, labeled as task (T2)). It is important to distinguish between these two kinds of regions when changing a feature. While developers can change regions labeled as (T1) without many concerns, regions labeled as (T2) require a more detailed inspection before changing the code, as the changes can break other unrelated functionality. The feature under consideration for this user study was Rhino's continuation context creation. This feature is responsible for creating a snapshot of a context, which contains the execution information needed to run Javascript code. One example of the information stored in a context is the call stack representation. These snapshots, which Rhino calls continuation objects, allow for users to pause execution and/or to return to a previous state in the execution.[3] A tutorial explaining the feature in detail was given to all participants,[4] which were given 20 minutes before the start of the experiment to study all materials provided. A time limit of 100 minutes was established to complete the task.

[3]More information about Rhino's continuation objects is available at `https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Rhino/New_in_Rhino_1.7R2`

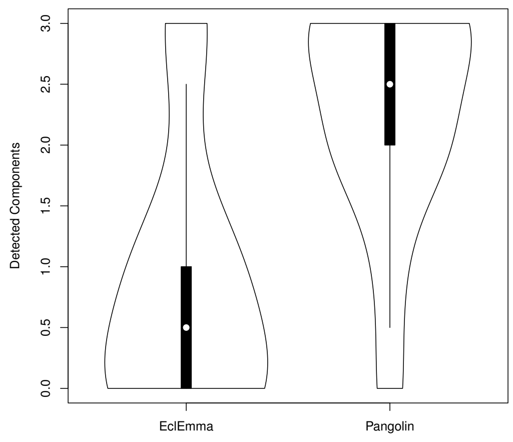
[4]All tutorials produced for this user study are available online at `http://gzoltar.com/pangolin/replication-package/`

The set of tests relevant to the creation of continuation contexts was gathered by the authors, manually inspecting every test before when setting up the experiment, and was given to all participants. The methodology for choosing the tests was as follows. First, all tests that did not execute Javascript scripts in interpreted mode were discarded. Rhino's continuations API is only supported in interpreted mode, which is selected by invoking `setOptimizationLevel(-1)` in the `Context` instance. Second, a further restriction taken into consideration is that continuation objects only capture contexts when scripts are called via the methods `executeScriptWithContinuations` and `callFunctionWithContinuations` from the `Context` instance. The search yielded two test classes, `ContinuationsApiTest` and `Bug482203Test`. The former exercises Rhino's continuations API, by executing scripts, pausing them, and capturing/restoring state; the latter was created to expose a bug in a previous version of Rhino that threw `NullPointerException` exceptions when capturing a continuation state. Manual inspection revealed that both test classes exercised the feature under consideration (namely, by covering the `captureContinuation` method in the `Context` class).

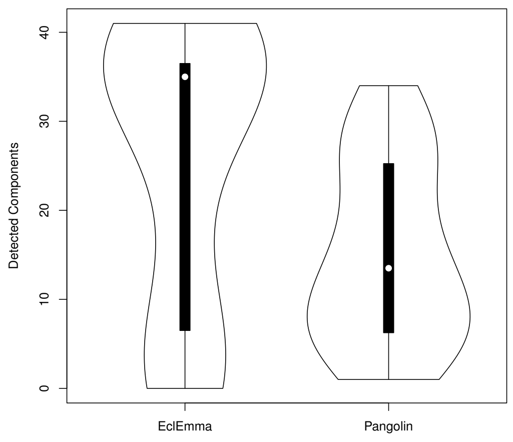
Participants were divided into two groups. One group comprising 26 pairs of participants was asked to use the PANGOLIN plugin to complete the task. As all participants were unfamiliar with PANGOLIN, a short tutorial explaining how to work with the tool (and how to interpret the results) was shown. For this group of participants to successfully complete the task, they need to use the tool to indicate the set of tests exercising the feature and run PANGOLIN's analysis. After the analysis is complete, the sunburst visualization appears in the corresponding Eclipse view. To identify the code regions that implement the feature (T1), participants should look for components whose association measure is 1 (color coded as red). Code regions shared among several features (T2) are components whose association measure is above 0 and below 1, and therefore color coded as different shades of yellow.

The other group of participants comprised by 28 pairs was the control group. Participants were asked to use the features from a standard version of the Eclipse IDE and its code-coverage plugin EclEmma[5], that shows, for a set of tests, what statements were executed. A short tutorial on how to work with EclEmma plugin was given beforehand. For the task to be successfully completed, participants need to gather the code-coverage information of all tests that exercise the feature, and compute their intersection. The intersection between these tests denotes the code regions that were executed on every test. A set difference between this intersection and the coverage of remaining tests in the test suite allows us to identify the regions that (T1) exclusively implement the feature and that (T2) are shared among many features, as is depicted in Figure 5.2.

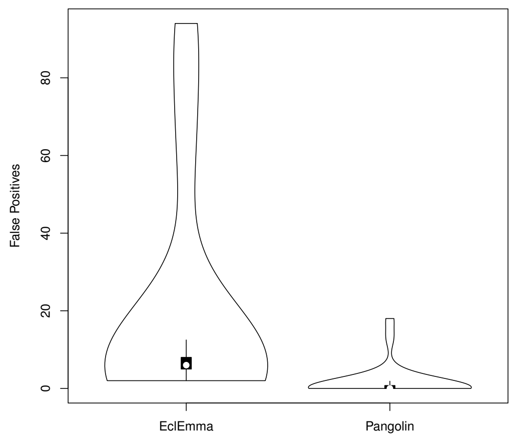
[5] Available at `http://www.eclemma.org/`



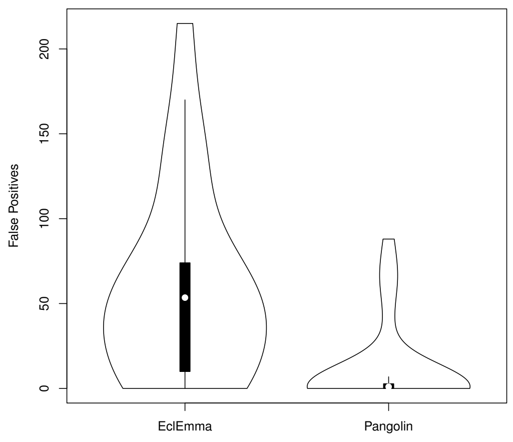
(a) Detected implementation components (T1).



(b) Detected shared components (T2).



(c) False positives labeled as implementation components (T1).



(d) False positives labeled as shared components (T2).

Figure 5.3: Violing plots depicting both groups' accuracy when labeling components.

For the particular feature considered in this user study, Rhino's continuation context creation (described in the previous subsection), users have to identify code regions in 3 classes that exclusively implement the feature (T1), and another 41 classes where that feature is used among many others (T2).

5.3.3 Results & Discussion

From the group that used the PANGOLIN plugin, participants were able to correctly identify a median of 2.5 classes from category (T1) and 13.5 from category (T2). In the group that used the code-coverage plugin EclEmma, participants identified a median of 0.5 classes from category (T1) and 35 from category (T2). Figures 5.3a and 5.3b show violin plots[6] depicting the amount of correct components detected by participants. We can see that, for identifying components in category (T1), participants working with PANGOLIN were able to achieve better results. In fact, over two thirds of the pairs of participants working with that plugin were able to find at

[6]A violin plot is the combination of a box plot and a kernel density plot.

Table 5.1: Mann-Whitney U tests.

Null Hypothesis	U	p-value	EclEmma Mean	PANGOLIN Mean	Effect size (d)
Detected components (T1) come from the same pop. (Figure 5.3a)	127.5	1.12×10^{-5}	0.82	2.27	Large (1.46)
Detected components (T2) come from the same pop. (Figure 5.3b)	223.5	7.60×10^{-3}	25.07	15.03	Medium (0.75)
False positives (T1) come from the same pop. (Figure 5.3c)	48.0	1.03×10^{-8}	19.14	1.65	Large (0.83)
False positives (T2) come from the same pop. (Figure 5.3d)	130.0	2.23×10^{-5}	57.93	10.88	Large (1.07)
Elapsed time comes from the same pop. (Figure 5.4)	234.0	1.20×10^{-2}	64.10	51.23	Medium (0.66)

least two correct components (out of three in total), as opposed to participants using EclEmma, where only 5 pairs identified at least two correct components.

As for the identification of components from category (T2), participants using EclEmma showed an increased overall accuracy when compared to PANGOLIN. However, and along with the correct code regions, there were several false positives identified by participants. While identifying regions that exclusively implement the feature (task category (T1)), the EclEmma group registered a median of 6 false positives, whereas the group using PANGOLIN registered a median of 0 false positives. The second category, code regions with shared features, yielded a median of 53.5 false positives when using EclEmma versus only 1 false positive when PANGOLIN is used. The amount of false positives each pair of participants identified is depicted in the violin plots from Figures 5.3c and 5.3d. In both categories, we see a substantial increase in the amount of false positives when the code-coverage EclEmma plugin is used to perform the requested task. This happens because the majority of participants using EclEmma, after gathering code-coverages for the indicted test cases, did not perform an intersection of the traces, as depicted in Figure 5.2. As a result, a considerable number of components were labeled incorrectly.

We also performed statistical tests to assess whether the gathered metrics yielded statistically significant results. The statistical test used is the Mann-Whitney U test [Mann and Whitney, 1947]. The reason we use Mann-Whitney instead of, *e.g.*, Student's t-test is because it does not assume that the data is normally distributed. In fact, a Shapiro-Wilk test [Shapiro and Wilk, 1965] on the data confirms that the distributions are not normal.

The results shown on Table 5.1 include the description of the null hypothesis, the task category from which the data originated, the test's U statistic and p-value. The p-values for all tests performed suggest that, according to the data, the null hypothesis must be rejected. The first four tests indicate that the two groups of participants can be considered statistically significant with 99% confidence, the last one has a 95% significance. Also shown are the means gathered for both groups (using EclEmma and using Pangolin) for each test. Lastly, the Cohen's difference between two means (d) for measuring the effect size is shown, as well as Cohen's qualitative effect size description [Cohen, 1988].

Revisiting the first research question:

Research Question 5.1

Can programmers using PANGOLIN pinpoint a feature more accurately than by inspecting standard test coverage traces?

Answer: Results show that the information about the program provided by the *SFC* analysis and the sunburst visualization is more accurate (particularly, in category (T1)) than requiring users to inspect and compare several traces with a code-coverage tool. Although in category (T2), users working with EclEmma were able to detect more components, this approach yielded a large number of false positives, which will most likely increase the comprehension effort, as users will need to inspect those components and deem then dissociated from the feature.

Task completion time for each pair of participants was also gathered. Although a time limit of 100 minutes was established, only two pairs required that amount to submit their results. Figure 5.4 depicts the elapsed time for the two groups of participants. Overall, the group using PANGOLIN completed the task in less time compared to the group using EclEmma. Participants using PANGOLIN took a median of 50 minutes to complete, whereas participants working with the EclEmma plugin took 60 minutes. The main reason for participants using EclEmma taking longer to complete the task is the fact that, after gathering the coverage information, they needed to perform the coverage analysis as shown in Figure 5.2. Participants using PANGOLIN only needed to gather the information shown to them via the sunburst visualization. No extra analysis was required.

Revisiting the second research question:

Research Question 5.2

Is the time taken to pinpoint a feature with PANGOLIN at least comparable to inspecting standard test coverage traces?

Answer: We conclude that using PANGOLIN does not negatively impact the time needed for comprehension processes, with the added advantage of being more accurate (as seen by answering **RQ5.1**).

5.3.4 Threats to Validity

Construct Validity Regarding construct validity, a threat is the fact that we measure components at a class level. Although the majority of the PANGOLIN groups have provided methods and line numbers in their reports, EclEmma groups had difficulty

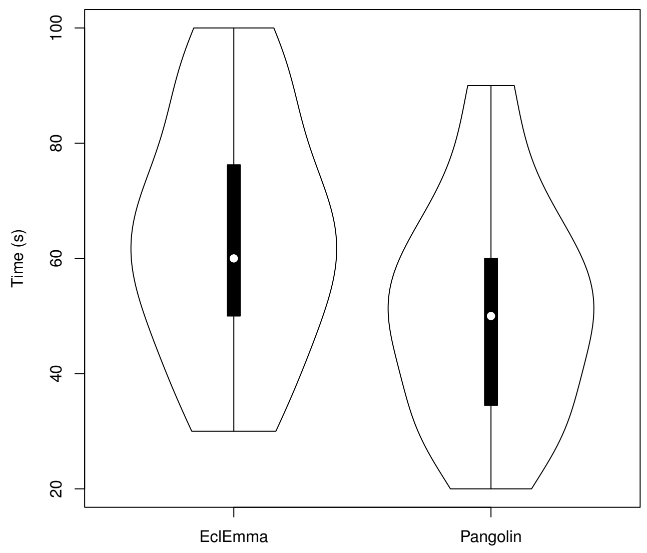


Figure 5.4: Time required by each pair of participants to complete the task, sorted by ascending order.

reporting components of finer granularity. Faced with this fact, we had to settle with class granularity so that both approaches could be compared.

Internal Validity A potential threat to the internal validity of this study is related to the selection of Mozilla Rhino as the application under analysis. When choosing the application for our study, our aim was to opt for application that (1) is indicative of a general, large-sized application being worked on by several people, and that (2) also provides significant understanding challenge to participants. To reduce selection bias, we decided to choose an application used in the evaluation of related work [Eaddy et al., 2008a; Eaddy et al., 2008b; Y. Zhang et al., 2006]. Other threat is the fact that the participants worked in pairs and chose who they wanted to pair up with within each group, which can introduce a potential bias to the experiment.

External Validity The main threat to the external validity of these results is the fact that participants were given the set of tests that exercise the feature under consideration. Information about what features each test is exercising may not be available or difficult to obtain. In fact, software projects may not even have tests for every feature (we attempt to mitigate this issue in Section 5.4). Also, all participants in the user study were software engineering students, and the study was performed in an academic setting, so it may not correctly reproduce the problems that the industry deals with. However, we argue that our setting closely resembles an important challenge faced in the software industry regarding program comprehension: introducing junior programmers into well established projects.

5.4 Participatory Feature Detection (*PFD*)

Feedback from the user study participants was that PANGOLIN had helped them locating the fault. However, at the end of the experiment, when participants were running the analysis against other projects, they were rarely able to successfully perform feature localization. This happens because the majority of the projects they experimented with did not have any unit tests, so the hit spectra matrix, required for the *SFC* analysis, was empty. In fact, as was pointed out in Section 5.3.4, our approach provides an automated way of locating features in the code, given that (1) there is a test suite and (2) the mapping between features and tests is known. In real software development scenarios, although it is good practice to test the system and to maintain a test-feature mapping, these are rarely available, which limits the applicability of the approach. To address this concern, we introduce and evaluate the concept of **Participatory Feature Detection (*PFD*)**.

5.4.1 The *PFD* Concept

The PANGOLIN tool has limited applicability when there is no pre-existing set of transactions (*i.e.*, no test suite available) or when there is no information on which runs have exercised the feature we are looking for. To address these issues, we extended both our *SFC* approach and our tool to account for user involvement. *PFD* allows users to capture manual interactions with the system, enabling them to label each interaction as associated or dissociated with the feature.

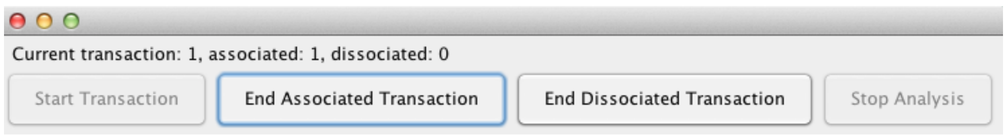

Current transaction: 1, associated: 1, dissociated: 0

Figure 5.5: PANGOLIN's *PFD* window.

As *SFC* only requires abstracted execution traces (*i.e.*, program spectra), the instrumentation code—to gather such traces—that is injected into the application under test causes minimal impact on performance. This means that PANGOLIN can be enhanced to feature user participation in an online fashion, where users are part of the analysis loop, receiving immediate feedback through the sunburst visualization after labeling each interaction. We have extended our tool to display an extra window during runtime, as shown in Figure 5.5, and the typical workflow of feature localization with *PFD* is as follows:

1. The analysis begins by running PANGOLIN. The subject application starts running and the *PFD* window appears (*cf.* Figure 5.5).
2. To begin a transaction, we click the 'Start Transaction' button.

3. PANGOLIN starts recording the trace of the application. We can now interact with the application and execute the feature we are trying to locate.
4. By pressing the 'End Associated Transaction' button, we are ending the current transaction, and labeling our interactions with the system as associated.
5. PANGOLIN runs the *SFC* analysis and displays the current results in its sunburst view. After one associated transaction, the visualization will look like Figure 5.6a, where every component in the trace has an association measure of 1.0.
6. After registering associated transactions, we need to capture dissociated interactions. To do so, we press the button 'Start Transaction', interact with the system without exercising the feature we are looking for, and then finish the transaction by pressing 'End Dissociated Transaction'.
7. When a new transaction is recorded, PANGOLIN will automatically update the SFC analysis and the sunburst visualization. Step 6 can be regarded as a way to minimize the slice of code that needs inspection, and can be repeated by registering different interactions with the system.

Figures 5.6a to 5.6d show the impact of adding new dissociated transactions to the analysis. By showing the updated report after every transaction, this enables the user to determine when to stop interacting with the system. It is worth noting that adding multiple associated transactions can also be beneficial for minimizing the slice of code to be inspected, since *SFC* will rank the intersected code regions (labeled as (T1) in Section 5.3.2) as more likely to contain the implementation of the feature.

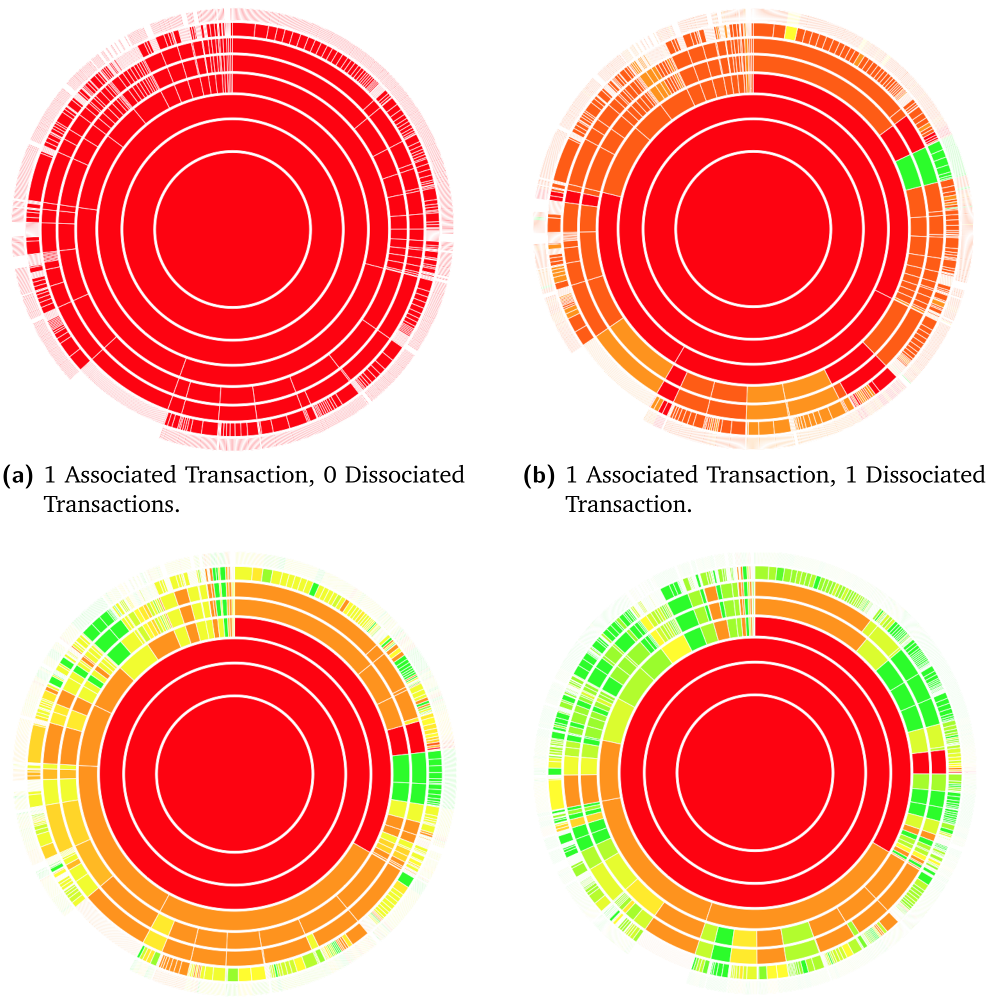
The *PFD* approach, since it requires user participation, is suited for analyzing interactive applications (in particular, **Graphical User Interface (*GUI*)** applications), as most of their features can be triggered by interacting with the interface in some fashion.

5.4.2 Case Study with JHotDraw

In order to evaluate the *PFD* approach, we performed a case study with the JHotDraw project. In this case study, we analyze what is the gain in information when adding associated and dissociated transactions to find a pre-selected feature.

JHotDraw[7] is a customizable Java framework for editing drawings. It can be used for sketches, diagrams, and artistic drawings. For the case study, we used the latest version available at the time of writing (JHotDraw 7, revision 789). It is comprised by 688 classes and 65 packages, with over 82,000 lines of code. As opposed to the Rhino project, used in our user study in Section 5.3, version 7 of JHotDraw does not

[7]Available at `http://sourceforge.net/projects/jhotdraw/`.


(a) 1 Associated Transaction, 0 Dissociated Transactions.

(b) 1 Associated Transaction, 1 Dissociated Transaction.

(c) 1 Associated Transaction, 2 Dissociated Transactions.

(d) 1 Associated Transaction, 3 Dissociated Transactions.

Figure 5.6: Sunburst report updated after each transaction.

currently contain any unit tests. Therefore, one could not use the non-*PFD* variant of PANGOLIN to locate its features.

The feature under consideration for this case study was the creation of triangle shapes in the drawing canvas. The aim of this exercise is to identify where that feature is implemented by just interacting with the application, and without having to inspect many spurious code locations. To do so, we recorded several associated transactions and dissociated transactions. Examples of associated transactions include creating a triangle shape of default size by single-clicking in the drawing area, creating a triangle with a custom size by click-dragging in the drawing area, and copy-pasting a triangle. Examples of dissociated transactions include changing an existing shape's color, resizing it or saving a drawing to a file.

With this case study, we aim to answer the following research questions:

Research Question 5.3

Are manual user interactions an accurate input to the *SFC* analysis when test cases are unavailable?

Research Question 5.4

What is the impact of incorrect interaction classifications on the accuracy of the *PFD* report?

In **RQ5.3** we want to assess if *PFD* can be considered as a fallback alternative in instances where there is no test suite available. **RQ5.4** is concerned with the viability of placing a human in the analysis loop, which consequently may lead to errors classifying system interactions.

5.4.3 Evaluation

We further detail the experiments performed to assess the accuracy of the participation-based extension of the *SFC* method. To evaluate our approach, we propose a metric that quantifies the accuracy of the report generated by *SFC*

$$\text{accuracy} = \frac{\sum_i^{R^+} \varepsilon(i) \cdot a_f(i)}{|R^+|} \tag{5.1}$$

where R^+ is the list of components that were scored with non-zero association measure. $\varepsilon(i)$ is a membership function that states whether component i is in fact part of the feature implementation. $\varepsilon(i)$ is given by

$$\varepsilon(i) = \begin{cases} 1 & \text{if component } i \text{ is part of the feature implementation} \\ -1 & \text{otherwise} \end{cases} \tag{5.2}$$

Accuracy can range between -1 and 1. In the best case scenario, where every reported component has a score of 1.0 and is a member of the feature implementation, the accuracy value would be 1. In the worst case scenario, the report would be solely comprised of dissociated components, and its accuracy would be -1.

The first experiment was set up in order to emulate the conventional use of *PFD*, where users would start by exercising the feature in the first transaction, and after that they would record several dissociated interactions with the aim of reducing their code inspection efforts. Therefore, in this experiment we analyze the impact of adding new dissociated transactions, given that the first recorded transaction is an associated one. The results are shown in Figure 5.7. As to be expected, the first transaction—the associated transaction where the feature is exercised—causes

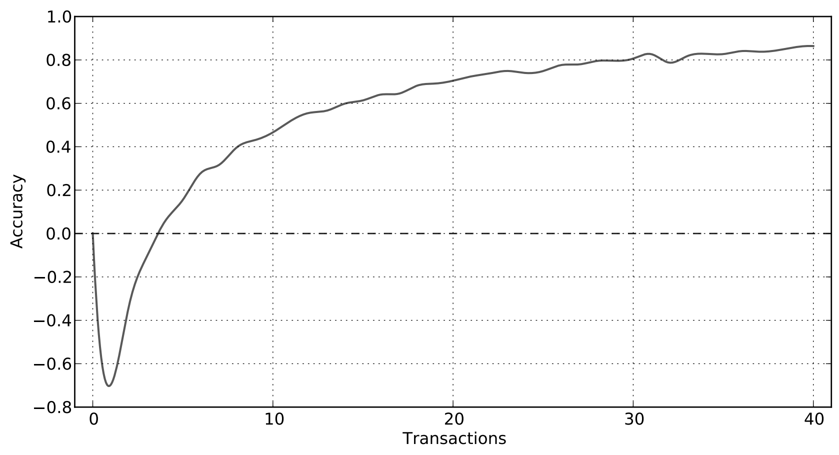


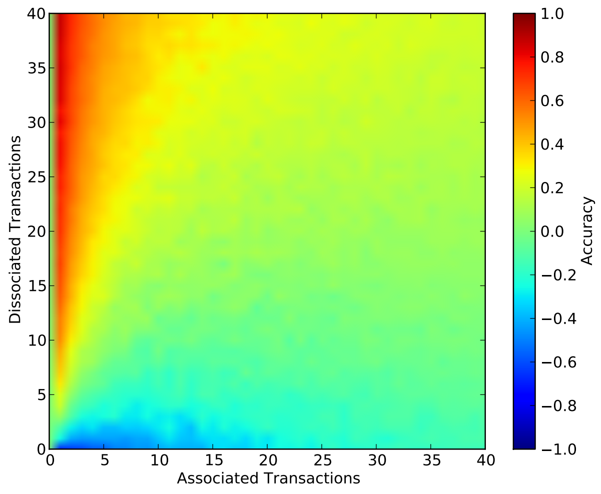
Figure 5.7: Accuracy metric for each recorded transaction.

a steep dip in the accuracy graph. Since at this time there is only one recorded execution and since there are typically many more dissociated components that are active in a single execution trace, this means that the accuracy will be negative. In the case of the triangle shape creation feature for JHotDraw, the accuracy drops to a value close to -0.7.

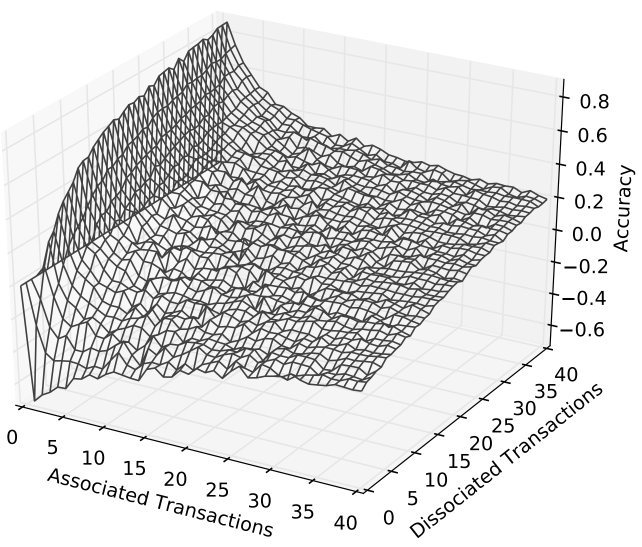
When dissociated transactions are added, we see a steady increase in the accuracy of the *SFC* analysis with *PFD*. This happens because, as different transaction traces are included in the spectrum-based analysis, we are reducing the cardinality of the set of components that are active exclusively in the associated transaction. This means that the association measure for dissociated components will decrease and, conversely, associated components will remain with a high association measure. We see that, by collecting a low amount of transactions (about 20 in the case of the triangle shape creation), we are able to achieve a high accuracy in the feature localization.

In the experiment above, we have only considered one transaction as associated, and analyzed the impact on accuracy of adding dissociated transactions. We have also performed another experiment where we vary the number of associated and dissociated transactions in the feature localization. Results are presented in Figure 5.8, under the format of a 3D plot (Figure 5.8b), and as an heat map (Figure 5.8a). We conclude that having more dissociated transactions yields a better accuracy than having more associated ones. This happens because, generally, two associated traces are much more similar than an associated trace and a dissociated trace. Therefore, a dissociated trace is better at exonerating components and reducing their score. This means that there is more information gained by adding a dissociated transaction than by adding associated transactions to the analysis.

Revisiting the third research question:



(a) Heat map.



(b) 3D plot.

Figure 5.8: Accuracy metric when changing the cardinality of associated and dissociated transactions.

Research Question 5.3

Are manual user interactions an accurate input to the *SFC* analysis when test cases are unavailable?

Answer: Allowing users to manually execute interactive applications and label their interactions as associated or dissociated can be an accurate alternative to the automated test suite. We found that a small number of transactions is enough to achieve considerable accuracy and that the approach is resilient to classification mistakes. We also found that the information gained by adding one dissociated transaction is higher than adding one associated transaction. Therefore, the best scenario to achieve accurate results with the participatory approach is to collect just a few associated transactions, and to collect as many dissociated transactions as possible.

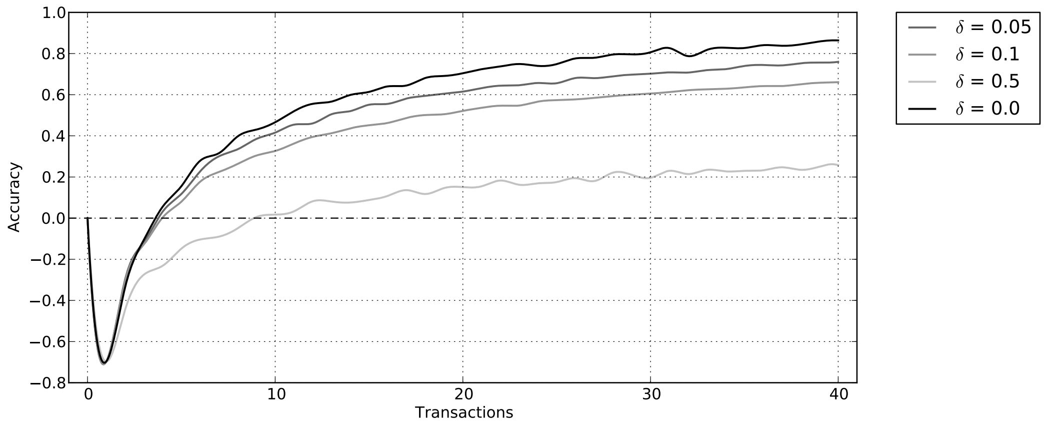


Figure 5.9: Impact of misclassification (δ) in *PFD*'s accuracy.

Since this participation-based approach to *SFC* involves users in the analysis loop and requires them to interact with the system, *PFD* has to be resilient to eventual mistakes when labeling executions. To assess how our approach behaves when there are erroneous transactions present, we consider the concept of misclassification (δ). δ is the probability that a transaction labeled as dissociated by the user is, in fact, associated with the feature. For that, we have repeated the first experiment with δ equal to 0.05, 0.10 and 0.50. The results are shown in Figure 5.9.

Revisiting the fourth research question:

Research Question 5.4

What is the impact of incorrect interaction classifications on the accuracy of the *PFD* report?

Answer: Both $\delta = 0.05$ and $\delta = 0.10$ show some decrease in accuracy, but they remain, however, comparable to having $\delta = 0.0$. In fact, we are able to achieve similar accuracies if we increase the number of dissociated transactions in both misclassification scenarios. This means that if users are unsure if some of the transactions they are collecting are in fact dissociated, then they should add more transactions to mitigate this effect. At the extreme case of $\delta = 0.50$, the results yielded by *SFC* are much more inaccurate. In such case, the score for the associated components decreases, because their activity pattern starts to differ from the (manually labeled) evolution vector.

5.4.4 Threats to validity

Construct Validity The case study was conducted by the authors. To ensure no bias in the execution of the experiment, one of the authors selected the feature. The other, which had no prior knowledge of JHotDraw's code, was responsible for running the PANGOLIN analysis. Misclassifications were labeled by checking each transaction's coverage and intersecting it with the feature implementation region.

Internal Validity A pre-requisite to using *PFD* is that the target applications need to have some kind of interactive interface so that the user can label different transactions (i.e., interactions). For instance, Rhino—the subject of our user study described in the previous section—, being a Javascript interpreter library for java, cannot be used in *PFD* mode, as it is not an interactive application. For this reason, we selected JHotDraw as the subject of this case study. This application has also been used in previous related work [Cornelissen et al., 2008]. One advantage of using JHotDraw is to show that a project with no test cases can be analyzed using *PFD*. In fact, this participatory approach is an attempt to increase the applicability domain of *SFC* by removing the requirement of having existing test cases. Instead, users are asked to interact with the application to generate coverage information that *SFC* can act upon.

External Validity The results presented may not generalize for all kinds of interactive applications. Due to the different ways applications can handle interaction events, classifying an interaction as associated/dissociated may not be trivial. Also, although it is shown that the approach is resilient with regards to varying levels of interaction misclassifications, there is, to our knowledge, no study on the actual value for the average misclassification rate for interactive applications.

5.5 Related Work

Various techniques and tools were developed as a result of several years of research into trace visualization and feature localization [Cornelissen et al., 2009; Dit et al., 2013]. This section provides an overview—not meant to be exhaustive—of the related work in this area.

Trace Visualization Pauw et al. [1993] developed a tool—coined Jinsight—for visually exploring a program's runtime behavior [Pauw et al., 1998]. Although this tool was shown to be useful for program comprehension, scalability concerns render the tool impractical for use in large applications. Reiss and Renieris [2001] showed that execution traces are typically too large to be visualized and understood by the user. As such, Reiss proposed a way to select and compact trace data to improve the visualization's intelligibility. Live run-time visualizations were also proposed as a way to reduce overheads [Reiss, 2003], but made it harder to visualize entire executions.

Ducasse et al. [2004] proposed a way of representing condensed runtime metrics (such as attribute-usage frequencies, object allocation frequencies, object lifetime, among others) with the use of *polymetric* views. Greevy et al. [2006] proposed a 3D visualization of the run-time traces of a software system. Components were depicted as towers whose height was influenced by the amount of runtime instances created. The main objective of this technique is to determine which system regions are involved in the execution of a certain feature, but the visualization may not be trivial to grasp (particularly in large applications).

Cornelissen et al. [2007] developed a tool—Extravis—that visualized execution traces by employing two synchronized views: a circular bundle view for structural elements and an interactive overview via a sequence view. Its effectiveness was also demonstrated for three reverse engineering contexts: exploratory program comprehension, feature detection and feature comprehension [Cornelissen et al., 2008]. Pinzger et al. [2008] proposed DA4Java, a tool that represented the source-code as a nested graph. Vertices in the graph represented code components, such as packages, classes and methods, and edges represented dependencies (*e.g.*, inheritance or method calls). Graph representations were also used in other works. Such is that of Yazdanshenas and Moonen [2012], which, like PANGOLIN, was able to visualize information flow at various abstraction levels. Ishio et al. [2012] also used graphs to generate interprocedural data-flow paths.

Trümper et al. [2013] implemented the TraceDiff tool, to ease the comparison of large-scale system traces. The tool provided visualizations featuring a modified hierarchical edge-bundling layout and icicle plot-node aggregation, to address scalability in large traces. Maletic et al. [2011] proposed the MosaiCode tool, that uses a 2D metaphor to support the visualization and understanding of various aspects of large scale software systems. It supported multiple coordinated views of these systems and leveraged a mosaic visualization to map their characteristics so that it is easy to understand by programmers, managers, and architects.

Stengel et al. [2011] developed the View Infinity tool. It provided a *zoomable* interface of **Software Product Lines (*SPL*)**. In *SPL*, software is implemented in terms of reusable user-visible characteristics. Features in *SPL* are particularly difficult to understand due to the inherent variability of this type of applications. Like PANGOLIN, the View Infinity tool offered a customizable granularity visualization, as well as a navigable interface.

Feature Localization Work related to locating features in code includes the software-reconnaissance approach proposed by Wilde and Scully [1995]. This approach tried to answer the question *"In which parts of this program is functionality*

X implemented?" using only dynamic information, namely execution traces. Similarly to *SFC*, the Software Reconnaissance approach distinguishes two sets of test cases (or scenarios): scenarios that activate the feature, and scenarios that do not activate the feature. The former are used to locate the portions of code that implement the feature and the latter are used to reduce the size of those code portions. The *SFC* approach differs from this approach as it uses a spectrum-based analysis to further indict or exonerate components based on each observed transaction. Another approach to feature localization, coined SNIAFL, was proposed in the work of Zhao et al. [2006]. This approach uses static techniques to locate features in the source code. The main idea behind the approach is to use information retrieval techniques to reveal the basic connection between features and computational units in source code, based on the premise that programmers use meaningful names as classifiers.

Information-foraging-based theories to explain information-seeking strategies have been used in the context of program comprehension and software engineering before. Relevant works include those of Ko et al. [2006] in the context of software maintenance; Romero et al. [2007], Lawrance et al. [2013], and Fleming et al. [2013], and Piorkowski et al. [2012] in software debugging; Chi et al. [2001] and Spool et al. [2004] for website design and evaluation.

5.6 Summary

In this chapter, we detailed the application of *SFL* techniques to the context of feature localization, as we initially suggested in Section 1.3.4. Concretely, in this chapter:

- We introduced *SFC*, which provides a *mapping* between similar concepts from fault localization concepts and feature localization (Section 5.2, page 93).
  - Instead of calculating the correlation to test failures, *SFC* calculates the correlation to feature usage patterns. *SFC* can thus rank components by their likelihood of being part of a feature's implementation, thereby reducing the effort of code inspection.
- We presented PANGOLIN, an Eclipse plugin that implements the *SFC* analysis for Java projects and presents tree-based radial visualization where each component is color-coded according to their likelihood of being associated with the feature being analyzed (Section 5.2.2, page 95).
- We conducted a user study to assess the effectiveness of *SFC* versus a coverage tool, where participants were asked to pinpoint the source-code location where a certain feature was implemented in the Rhino project (Section 5.3, page 96). Results show that:
  1. *SFC* users were able to pinpoint associated components more accurately (answering **RQ5.1**, page 102); and

2. *SFC* users completed the feature localization task in less time compared to the group using a coverage tool (answering **RQ5.2**, page 102).

- We extended the *SFC* approach to allow for user participation in interactive applications (Section 5.4, page 104). With the *PFD* approach, users can capture manual executions of the application and label them as associated or dissociated with the feature. This allows the use of feature localization for interactive applications, even if there are no automated tests available. Our evaluation of *PFD* shows that:
  1. Allowing users to manually execute interactive applications and label their interactions as associated or dissociated can be an accurate alternative to the automated test suite (answering **RQ5.3**, page 109); and that
  2. While misclassification impacts accuracy, it can be mitigated by increasing the number of recorded transactions (answering **RQ5.4**, page 110).

6 Conclusions

The key to understanding complicated things is knowing what not to look at.

— **Gerald Jay Sussman**

Throughout this thesis, we have detailed the inner workings of *SFL* techniques, outlined some of the flaws and shortcomings hindering them from widespread adoption and use in practice, and proposed several methodologies to improve their applicability and effectiveness, motivated by the question we posed at the beginning of the thesis (*cf*. Section 1.3):

Can we improve the usefulness and effectiveness of spectrum-based analyses throughout the software development life cycle?

The question above has spawned four main avenues of research covered in this thesis—which we denominated by research goals, and introduced in Section 1.3—focused on improving different aspects of spectrum-based analyses throughout the software development life cycle. We start this concluding chapter by outlining the work performed in the scope of each research goal. Then, we list the main contributions resulting from this work. Finally, we discuss recommendations for future research.

6.1 Research Goals

Research Goal 1

Can we create a near-optimal metric to assess *SFL* diagnosability of test suites while avoiding the assumptions held in previous work?

This research goal highlights the importance of diagnosability—the ability to effectively locate potential faults in the code—as a criterion for assessing the quality of a test suite. In Chapter 2, we propose *DDU* as a measurement of program spectra diagnosability. Ideal diagnostic ability can be proved to exist when a suite reaches maximum entropy, however, the number of tests required to achieve that is impractical, as the number of components in the system increases. *DDU* focuses on three particular properties of entropy: (1) ensures that test cases are diverse; (2) ensures that there are no ambiguous components; (3) ensures that there is a proportional number of tests of distinct granularity; while still ensuring tractability. As opposed to adequacy measurements such as coverage, which mainly tackle the issue of error detection, a diagnosability measurement like *DDU* analyses how combinations of components are exercised in tandem in order to maximize the usefulness of fault localization techniques at pinpointing the *cause* of any error that may occur.

Our topology-based simulation of program spectra was able to reveal that *DDU* effectively establishes an upper-bound on the maximal effort required to diagnose faults, regardless of fault type or cardinality. We also performed an empirical evaluation to assess *DDU* as a metric for diagnosability. Our experiments used the EVOSUITE tool to generate test suites for faulty programs from the DEFECTS4J catalog, optimizing different metrics. We observed a statistically significant increase in diagnostic performance (of about 34%) when locating faults by optimizing *DDU* compared to branch-coverage.

Research Goal 2

What is the prevalence of single-fault fixes versus multiple-fault ones, and what is their impact in similarity-based *SFL*?

In Chapter 3, we study the prevalence of single-fault fixes in open-source Java projects, motivated by the fact that fault predictors (used by similarity-based *SFL* approaches) can perform optimally in the event of a system being single-faulted, with minimal computational overhead. Our hypothesis is that while a software application can have many dormant bugs, these bugs are detected (and fixed) individually, thus constituting single-faulted events.

We describe an approach for mining software repositories in search for fixes—*i.e.*, source code modifications that eliminate faults from the system—and for classifying said fixes according to the number of bugs they eliminate. The motivation behind creating such methodology is to study how debugging actually happens in practice and whether there is prevalence of single-fault fixes throughout the development of software.

We conducted an experiment with hundreds of open-source Java projects from GitHub, mining their repositories and cataloging the identified fixes. Overall, we have found 1375 fixes in over 70 projects. Out of all fixes, 82.5% were single-faulted (*i.e.*, only eliminate one bug from the system), indicating that single-faults are indeed prevalent among fixes. We have found that, for the detected single-faults, fault predictors O and O^P manage to achieve high diagnostic accuracy. For the remaining multiple-faulted scenarios, average-fault and worst-fault diagnostic accuracies decreased using most fault predictors. A glaring exception is with the O predictor: because its definition assumes systems are single-faulted, it exhibits an essentially random diagnostic performance in multiple-faulted scenarios.

Research Goal 3

Can we augment spectra with qualitative, contextual information to improve *SFL* diagnoses?

In Chapter 4, we propose *Q-SFL*, a new approach to spectrum-based fault localization that leverages *QR*. *QR* is a research field of Artificial Intelligence focused on the study of ways to describe continuous variables by a set of finite qualitative states, allowing for the modeling, simulation and reasoning of complex systems. *Q-SFL* splits data variables from the software system under analysis into qualitative states through the creation of qualitative landmarks that partition the variables domain. These qualitative states are then considered as *SFL* components to be ranked using traditional fault-localization methodologies. Since we treat qualitative descriptions of variable domain partitions as components, our diagnostic reports not only recommend likely fault locations, but also what *behaviors* neighboring variables assume when failures are detected, facilitating the comprehension of the fault.

We evaluate the approach on subjects from the DEFECTS4J catalog of real faults from medium and large-sized open source projects. Results show that spectra which were *augmented* using qualitative partitioning of method parameters shows a (statistically significant) improvement in the diagnostic accuracy. However, no automated *black-box* partitioning strategy used in our evaluation was consistently better than the original spectra, meaning that more intricate strategies (possibly with access to source code) will likely be necessary for practical applications of the approach.

Research Goal 4

Can we leverage *SFL* for feature detection?

Chapter 5 tackles the challenge of understanding and locating code features pre-existing source code by leveraging concepts and algorithms from the field of software fault localization. We provide a mapping between the problem of fault localization and the problem of locating features in the code, and present an approach, coined *SFC*. The main difference between the two approaches is that, instead of calculating the correlation to failure patterns, *SFC* calculates the correlation to feature usage patterns. *SFC* can thus rank components by their likelihood of being part of a feature's implementation, thereby reducing the effort of code inspection.

The chapter also presents PANGOLIN, an Eclipse plugin that implements the *SFC* analysis for Java projects. This tool, besides running test cases and performing *SFC*, also presents the analysis by means of a tree-based code visualization called Sunburst, where each component is color-coded according to their likelihood of being associated with the feature being analyzed.

To assess the effectiveness of *SFC* and PANGOLIN, a user study was carried out, where participants where asked to pinpoint components that need to be inspected when evolving/changing a certain feature in the Rhino open source project. Results show that participants using PANGOLIN were able to more accurately pinpoint code that implemented the feature and code that used it when compared to participants only using test coverage reports.

We extend *SFC* to enable user participation in the analysis process. With the *PFD* approach users can, instead of using test cases, capture manual executions of the application and label them as associated or dissociated with the feature. This allows the use of feature localization in interactive applications, even if no automated tests are made available. Our experimental results lead us to conclude that users should strive to collect few associated executions and several dissociated executions to achieve considerable feature localization accuracy, and that the approach is resilient to misclassifications by the user.

6.2 Contributions

This thesis' main contributions are the following:

- Introduces *DDU*, a new metric to assess a test-suite's diagnostic ability to pinpoint a fault in the system using spectrum-based techniques. The *DDU* metric, intended to complement adequacy measurements such as branch coverage, was shown to be more accurate at assessing diagnosability than the state-of-the-art.

- Details a methodology for finding bug fixes in a software repository and labeling them according to their bug cardinality, based on a spectrum-based analysis. This methodology allows us to assess the prevalence of single-fault fixes throughout development and also to assess what is the difference in similarity-based *SFL*'s diagnostic accuracy when debugging single faults versus multiple faults.

- Outlines an extension to *SFL* that leverages *QR* to augment program spectra. This augmentation is done by partitioning system variables' runtime values into sets of qualitative descriptions, which are then treated as *SFL* components to be analyzed. With more contextual information encoded in the spectra, it allows for improved fault isolation and fault comprehension.

- Details an approach, *SFC*, that uses spectrum-based analysis to help software engineers in locating where specific features are implemented in the code and understanding how a program is structured. We consider this approach particularly useful in program maintenance and evolution tasks.

6.3 Recommendations for Future Research

Diagnosable Architectures The main purpose behind introducing the *DDU* metric in Chapter 2 was to evaluate a test suite's ability to pinpoint faults in the event of failures. This metric is therefore a fitting complement to test adequacy measurements, aimed at assessing the quality of test suites. However, the structure of the system under test (its *topology*, as we refer to in Chapter 2) will also necessarily influence its propensity for fault isolation. Hence, we consider the study of diagnosable architectures to be an important avenue of future work. Namely, we envision the creation of architectural design patterns, coding frameworks, runtimes, and even high-level languages in which diagnosability (and by extension, the ease of improving *DDU*) is one of the core concerns. Examples of existing research that could yield advancements in diagnosable architectures include:

- Model-driven software engineering [Schmidt, 2006] approaches to system design and verification. In particular, developments in both model-based test generation [Dalal et al., 1999] and model-based diagnosis techniques [Ye and Dague, 2010; Mayer and Stumptner, 2008] can help elicit precise design requirements for both testable and diagnosable systems;

- Automated invariant inference, both at runtime (*e.g.*, by training code *screeners* [Abreu et al., 2008]), and even at compile time (*e.g.*, by ensuring that data ownership and lifetime are explicitly declared and checked [Jung et al., 2018]) may help developers improve their understanding of the faulty behavior (and may also play a role as *Q-SFL* landmarking strategies);

- *"Pseudoexhaustive"* and (semi-)automated combinatorial testing [Kuhn et al., 2004; Kuhn et al., 2009] for generating thorough and diverse test suites (for instance, optimizing measurements such as *DDU* by studying the target's topology).

Differential Diagnosis As discussed in Chapter 3, since often times observed errors are due to single-faults, similarity-based *SFL* approaches—which are considerably less computationally demanding than reasoning-based counterparts—can be effectively employed with negligible overhead, improving the perceived user experience. In the event of multiple or intermittent faults, however, similarity-based *SFL* may be less accurate. Therefore, we envision that performance improvements in reasoning-based *SFL* may reduce the computational gap between the two kinds of *SFL* approaches and thus also reduce the incentive for using similarity-based techniques. We consider work (1) that takes into account previous diagnoses (such as the component goodness estimation approach by Cardoso and Abreu [2013a]), (2) that parallelizes the work required to perform the candidate generation step [Cardoso and Abreu, 2013b], and (3) that leverages information from the system's repository and bug tracker [Elmishali et al., 2016], to be examples of preliminary work in this direction. Expanding on the aforementioned work, the development of intermediate representations for storing set-cover constraints (akin to current strategies for solving online set-cover problems [Alon et al., 2009]) which only need to update the constraints of modified components between two code revisions, may substantially improve the candidate generation performance.

Automated Landmarking for Complex Types The *Q-SFL* approach outlined in Chapter 4 was shown to produce better diagnoses, even tough only primitive types (and object nullity) were considered for qualitative landmarking. Due to this limitation, no general landmarking strategy was able to achieve consistent results when compared to the baseline. Particularly, faults related to complex types will likely not be isolated. It this therefore essential to also partition and landmark complex data types, so that more types of faults can be covered. Hence, we believe future work focusing on the creation of automated partitioning strategies for these data types to be the next step in the development of *Q-SFL*. We envision that such partitioning strategies will need to adopt concepts and techniques from static and dynamic software analysis. Among other approaches, we consider the use of abstract interpretation [P. Cousot and R. Cousot, 2014] and delta debugging [Zeller, 1999], to be particularly suited for discriminating the qualitative states of a custom data type in an automated fashion.

Code Summarization The *SFC* approach (described in Chapter 5) allows users to leverage *SFL* techniques in the context of feature maintenance and evolution. We consider the *PFD* extension to be particularly useful for end-users to pinpoint feature implementations as they receive iterative, visual feedback while recording their interactions with the system. As future work, we plan to augment the information displayed to users, which currently only includes each component's association measures. One way to achieve this is by enhancing highly associated components with summaries of what they are responsible for, via code summarization techniques based on stereotypes [Moreno et al., 2013; Alhindawi et al., 2013], along with searchable, interactive visualization filtering options.

Bibliography

Abreu, Rui, Jácome Cunha, João Paulo Fernandes, Pedro Martins, Alexandre Perez, and João Saraiva (2014). “Smelling Faults in Spreadsheets”. In: *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*, pp. 111–120 (Cited on page 11).

Abreu, Rui and Arjan J. C. van Gemund (2009). “A Low-Cost Approximate Minimal Hitting Set Algorithm and its Application to Model-Based Diagnosis”. In: *Eighth Symposium on Abstraction, Reformulation, and Approximation, SARA 2009, Lake Arrowhead, California, USA, 8-10 August 2009* (Cited on pages 7, 52, 55).

Abreu, Rui, Alberto González-Sanchez, and Arjan J. C. van Gemund (2011). “A Diagnostic Reasoning Approach to Defect Prediction”. In: *Modern Approaches in Applied Intelligence - 24th International Conference on Industrial Engineering and Other Applications of Applied Intelligent Systems, IEA/AIE 2011, Syracuse, NY, USA, June 28 - July 1, 2011, Proceedings, Part II*, pp. 416–425 (Cited on page 26).

Abreu, Rui, Alberto González-Sanchez, Peter Zoeteweij, and Arjan J. C. van Gemund (2008). “Automatic software fault localization using generic program invariants”. In: *Proceedings of the 2008 ACM Symposium on Applied Computing (SAC), Fortaleza, Ceara, Brazil, March 16-20, 2008*, pp. 712–717 (Cited on page 119).

Abreu, Rui, Wolfgang Mayer, Markus Stumptner, and Arjan J. C. van Gemund (2009a). “Refining spectrum-based fault localization rankings”. In: *Proceedings of the 2009 ACM Symposium on Applied Computing (SAC), Honolulu, Hawaii, USA, March 9-12, 2009*, pp. 409–414 (Cited on page 10).

Abreu, Rui, Peter Zoeteweij, and Arjan J. C. van Gemund (2006). “An Evaluation of Similarity Coefficients for Software Fault Localization”. In: *12th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC 2006), 18-20 December, 2006, University of California, Riverside, USA*, pp. 39–46 (Cited on pages 6, 7).

– (2009b). “A New Bayesian Approach to Multiple Intermittent Fault Diagnosis”. In: *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, pp. 653–658 (Cited on pages 9, 36).

Abreu, Rui, Peter Zoeteweij, and Arjan J. C. van Gemund (2009c). "Spectrum-Based Multiple Fault Localization". In: *ASE 2009, 24th IEEE/ACM International Conference on Automated Software Engineering, Auckland, New Zealand, November 16-20, 2009*, pp. 88–99 (Cited on pages 6, 7, 10, 13, 28, 47).

Abreu, Rui, Peter Zoeteweij, Rob Golsteijn, and Arjan J. C. van Gemund (2009d). "A practical evaluation of spectrum-based fault localization". In: *Journal of Systems and Software* 82.11, pp. 1780–1792 (Cited on pages 3, 4, 92).

Alhindawi, Nouh, Natalia Dragan, Michael L. Collard, and Jonathan I. Maletic (2013). "Improving Feature Location by Enhancing Source Code with Stereotypes". In: *2013 IEEE International Conference on Software Maintenance, Eindhoven, The Netherlands, September 22-28, 2013*, pp. 300–309 (Cited on page 120).

Alipour, Mohammad Amin, August Shi, Rahul Gopinath, Darko Marinov, and Alex Groce (2016). "Evaluating Non-adequate Test-Case Reduction". In: *31st IEEE/ACM International Conference on Automated Software Engineering (ASE)* (Cited on page 45).

Alon, Noga, Baruch Awerbuch, Yossi Azar, Niv Buchbinder, and Joseph Naor (2009). "The Online Set Cover Problem". In: *SIAM J. Comput.* 39.2, pp. 361–370 (Cited on page 120).

Apel, Sven, Don Batory, Christian Kästner, and Gunter Saake (2013). *Feature-Oriented Software Product Lines*. Springer (Cited on page 16).

Artzi, Shay, Julian Dolby, Frank Tip, and Marco Pistoia (2010). "Directed test generation for effective fault localization". In: *Proceedings of the Nineteenth International Symposium on Software Testing and Analysis, ISSTA 2010, Trento, Italy, July 12-16, 2010*, pp. 49–60 (Cited on page 45).

Avižienis, Algirdas, Jean-Claude Laprie, Brian Randell, and Carl E. Landwehr (2004). "Basic Concepts and Taxonomy of Dependable and Secure Computing". In: *IEEE Transactions on Dependable Secure Computing* 1.1, pp. 11–33 (Cited on page 4).

Basili, Victor R. and Barry T. Perricone (1984). "Software Errors and Complexity: An Empirical Investigation". In: *Communications of the ACM* 27.1, pp. 42–52 (Cited on page 14).

Baudry, Benoit, Franck Fleurey, and Yves Le Traon (2006). "Improving test suites for efficient fault localization". In: *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*, pp. 82–91 (Cited on pages 12, 13, 20, 28).

Bettenburg, Nicolas, Meiyappan Nagappan, and Ahmed E. Hassan (2012). "Think locally, act globally: Improving defect and effort prediction models". In: *9th IEEE Working Conference of Mining Software Repositories, MSR 2012, June 2-3, 2012, Zurich, Switzerland*, pp. 60–69 (Cited on pages 64, 87).

Binder, Robert V. (2000). *Testing Object-oriented Systems: Models, Patterns, and Tools*. Object Technology Series. Addison-Wesley (Cited on page 43).

Bland, Mike (2014). “Finding More Than One Worm in the Apple”. In: *ACM Queue* 12.5, pp. 10–21 (Cited on page 2).

Böhme, Marcel and Abhik Roychoudhury (2014). “CoREBench: studying complexity of regression errors”. In: *International Symposium on Software Testing and Analysis, ISSTA ’14, San Jose, CA, USA - July 21 - 26, 2014*, pp. 105–115 (Cited on pages 48, 64).

Budd, Timothy Alan (1980). “Mutation Analysis of Program Test Data”. PhD thesis. Yale University New Haven CT USA (Cited on page 19).

Campos, José, Rui Abreu, Gordon Fraser, and Marcelo d’Amorim (2013). “Entropy-based test generation for improved fault localization”. In: *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, pp. 257–267 (Cited on pages 10, 36, 37, 42, 45, 85).

Campos, José, André Riboira, Alexandre Perez, and Rui Abreu (2012). “GZoltar: an eclipse plug-in for testing and debugging”. In: *IEEE/ACM International Conference on Automated Software Engineering, ASE’12, Essen, Germany, September 3-7, 2012*, pp. 378–381 (Cited on pages 55, 79, 87).

Cardoso, Nuno and Rui Abreu (2013a). “A Kernel Density Estimate-Based Approach to Component Goodness Modeling”. In: *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence, July 14-18, 2013, Bellevue, Washington, USA.* (Cited on pages 14, 28, 64, 87, 119).

– (2013b). “MHS2: A Map-Reduce Heuristic-Driven Minimal Hitting Set Search Algorithm”. In: *Multicore Software Engineering, Performance, and Tools - International Conference, MUSEPAT 2013, St. Petersburg, Russia, August 19-20, 2013. Proceedings*, pp. 25–36 (Cited on pages 7, 119).

Carey, John, Neil Gross, Marcia Stepanek, and Otis Port (1999). “Software Hell”. In: *Business Week*, pp. 391–411 (Cited on page 8).

Chi, Ed Huai-hsin, Peter Pirolli, Kim Chen, and James E. Pitkow (2001). “Using information scent to model user information needs and actions and the Web”. In: *Proceedings of the CHI 2001 Conference on Human Factors in Computing Systems, Seattle, WA, USA, March 31 - April 5, 2001.* Pp. 490–497 (Cited on page 113).

Chilenski, John Joseph and Steven P. Miller (1994). “Applicability of modified condition/decision coverage to software testing”. In: *Software Engineering Journal* 9.5, pp. 193–200 (Cited on page 19).

Cohen, Jacob (1988). *Statistical Power Analysis for the Behavioral Sciences*. Lawrence Erlbaum Associates (Cited on page 102).

Corbi, T. A. (1989). “Program understanding: Challenge for the 1990’s”. In: *IBM Systems Journal* 28.2, pp. 294–306 (Cited on pages 91, 92).

Cornelissen, Bas, Danny Holten, Andy Zaidman, Leon Moonen, Jarke J. van Wijk, and Arie van Deursen (2007). “Understanding Execution Traces Using Massive Sequence and Circular Bundle Views”. In: *15th International Conference on Program Comprehension (ICPC 2007), June 26-29, 2007, Banff, Alberta, Canada*, pp. 49–58 (Cited on page 112).

Cornelissen, Bas, Andy Zaidman, and Arie van Deursen (2011). “A Controlled Experiment for Program Comprehension through Trace Visualization”. In: *IEEE Trans. Software Eng.* 37.3, pp. 341–355 (Cited on page 44).

Cornelissen, Bas, Andy Zaidman, Arie van Deursen, Leon Moonen, and Rainer Koschke (2009). “A Systematic Survey of Program Comprehension through Dynamic Analysis”. In: *IEEE Transactions on Software Engineering* 35.5, pp. 684–702 (Cited on page 111).

Cornelissen, Bas, Andy Zaidman, Danny Holten, Leon Moonen, Arie van Deursen, and Jarke J. van Wijk (2008). “Execution trace analysis through massive sequence and circular bundle views”. In: *Journal of Systems and Software* 81.12, pp. 2252–2268 (Cited on pages 111, 112).

Cousot, Patrick and Radhia Cousot (1977). “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints”. In: *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pp. 238–252 (Cited on page 86).

– (2014). “Abstract interpretation: past, present and future”. In: *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS ’14, Vienna, Austria, July 14 - 18, 2014*, 2:1–2:10 (Cited on page 120).

D’Ambros, Marco, Michele Lanza, and Romain Robbes (2010). “An extensive comparison of bug prediction approaches”. In: *Proceedings of the 7th International Working Conference on Mining Software Repositories, MSR 2010 (Co-located with ICSE), Cape Town, South Africa, May 2-3, 2010, Proceedings*, pp. 31–41 (Cited on page 87).

Dalal, Siddhartha R., Ashish Jain, Nachimuthu Karunanithi, J. M. Leaton, Christopher M. Lott, Gardner C. Patton, and Bruce M. Horowitz (1999). “Model-Based Testing in Practice”. In: *Proceedings of the 1999 International Conference on Software Engineering, ICSE’ 99, Los Angeles, CA, USA, May 16-22, 1999.* Pp. 285–294 (Cited on page 119).

Dallmeier, Valentin and Thomas Zimmermann (2007). “Extraction of bug localization benchmarks from history”. In: *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), November 5-9, 2007, Atlanta, Georgia, USA*, pp. 433–436 (Cited on pages 48, 64).

DiGiuseppe, Nicholas and James A. Jones (2011). “On the influence of multiple faults on coverage-based fault localization”. In: *Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17-21, 2011*, pp. 210–220 (Cited on page 64).

Dit, Bogdan, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk (2013). “Feature location in source code: a taxonomy and survey”. In: *Journal of Software: Evolution and Process* 25.1, pp. 53–95 (Cited on page 111).

Dowson, Mark (1997). “The Ariane 5 software failure”. In: *SIGSOFT Software Engineering Notes* 22.2, p. 84 (Cited on page 2).

Ducasse, S., M. Lanza, and R. Bertuli (2004). “High-Level Polymetric Views of Condensed Run-time Information”. In: *Proceedings of the Conference on Software Maintenance and Reengineering (CSMR’04)*, pp. 309–318 (Cited on page 112).

Durumeric, Zakir et al. (2014). “The Matter of Heartbleed”. In: *Proceedings of the 2014 Internet Measurement Conference, IMC 2014, Vancouver, BC, Canada, November 5-7, 2014*, pp. 475–488 (Cited on page 3).

Eaddy, Marc, Alfred V. Aho, Giuliano Antoniol, and Yann-Gaël Guéhéneuc (2008a). “CERBERUS: Tracing Requirements to Source Code Using Information Retrieval, Dynamic Analysis, and Program Analysis”. In: *Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension*. ICPC ’08, pp. 53–62 (Cited on page 103).

Eaddy, Marc, Thomas Zimmermann, Kaitin D. Sherwood, Vibhav Garg, Gail C. Murphy, Nachiappan Nagappan, and Alfred V. Aho (2008b). “Do Crosscutting Concerns Cause Defects?” In: *IEEE Transactions on Software Engineering* 34.4, pp. 497–515 (Cited on page 103).

Elmishali, Amir, Roni Stern, and Meir Kalech (2016). “Data-Augmented Software Diagnosis”. In: *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA.* Pp. 4003–4009 (Cited on pages 14, 64, 87, 120).

Feldman, Alexander, Gregory M. Provan, and Arjan J. C. van Gemund (2008). “Computing Minimal Diagnoses by Greedy Stochastic Search”. In: *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008*, pp. 911–918 (Cited on pages 7, 52).

Fleming, Scott D., Christopher Scaffidi, David Piorkowski, Margaret M. Burnett, Rachel K. E. Bellamy, Joseph Lawrance, and Irwin Kwan (2013). “An Information Foraging Theory Perspective on Tools for Debugging, Refactoring, and Reuse Tasks”. In: *ACM Transactions on Software Engineering and Methodology* 22.2, 14:1–14:41 (Cited on pages 96, 113).

Forbus, Kenneth D. (1997). “Qualitative Reasoning”. In: *The Computer Science and Engineering Handbook*, pp. 715–733 (Cited on pages 68, 69).

Fraser, Gordon and Andrea Arcuri (2011). “EvoSuite: automatic test suite generation for object-oriented software”. In: *SIGSOFT/FSE’11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC’11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*, pp. 416–419 (Cited on page 35).

Gall, Harald C., Karin Hajek, and Mehdi Jazayeri (1998). “Detection of Logical Coupling Based on Product Release History”. In: *1998 International Conference on Software Maintenance, ICSM 1998, Bethesda, Maryland, USA, November 16-19, 1998*, pp. 190–197 (Cited on page 64).

Gharehyazie, Mohammad, Baishakhi Ray, and Vladimir Filkov (2017). “Some from here, some from there: cross-project code reuse in GitHub”. In: *Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, May 20-28, 2017*, pp. 291–301 (Cited on page 64).

Gong, Liang, David Lo, Lingxiao Jiang, and Hongyu Zhang (2012). “Diversity maximization speedup for fault localization”. In: *IEEE/ACM International Conference on Automated Software Engineering, ASE’12, Essen, Germany, September 3-7, 2012*, pp. 30–39 (Cited on page 45).

González-Sanchez, Alberto, Rui Abreu, Hans-Gerhard Gross, and Arjan J. C. van Gemund (2011a). “Prioritizing tests for fault localization through ambiguity group reduction”. In: *26th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, pp. 83–92 (Cited on page 12).

González-Sanchez, Alberto, Hans-Gerhard Gross, and Arjan J. C. van Gemund (2011b). “Modeling the Diagnostic Efficiency of Regression Test Suites”. In: *4th IEEE International Conference on Software Testing, Verification and Validation (ICST), Workshop Proceedings*, pp. 634–643 (Cited on pages 12, 20, 24, 28, 36).

Gousios, Georgios and Andy Zaidman (2014). “A Dataset for Pull-based Development Research”. In: *Proceedings of the 11th Working Conference on Mining Software Repositories*. MSR 2014, pp. 368–371 (Cited on pages 54, 63).

Gouveia, Carlos, José Campos, and Rui Abreu (2013). “Using HTML5 visualizations in software fault localization”. In: *2013 First IEEE Working Conference on Software Visualization (VISSOFT), Eindhoven, The Netherlands, September 27-28, 2013*, pp. 1–10 (Cited on pages 67, 87, 96).

Greevy, Orla, Michele Lanza, and Christoph Wysseier (2006). “Visualizing live software systems in 3D”. In: *Proceedings of the ACM 2006 Symposium on Software Visualization, Brighton, UK, September 4-5, 2006*, pp. 47–56 (Cited on pages 92, 112).

Hailpern, Brent and Padmanabhan Santhanam (2002). “Software debugging, testing, and verification”. In: *IBM Systems Journal* 41.1, pp. 4–12 (Cited on page 2).

Han, Jiawei, Jian Pei, and Micheline Kamber (2011). *Data Mining: Concepts and Techniques*. Elsevier (Cited on page 80).

Harrold, Mary Jean, Gregg Rothermel, Rui Wu, and Liu Yi (1998). “An Empirical Investigation of Program Spectra”. In: *Proceedings of the SIGPLAN/SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE ’98, Montreal, Canada, June 16, 1998*, pp. 83–90 (Cited on pages 3, 5).

Hartmann, Klaas, Dennis Wong, and Tanja Stadler (2010). “Sampling Trees from Evolutionary Models”. In: *Systematic Biology* 59.4, pp. 465–476 (Cited on page 29).

Herzig, Kim and Andreas Zeller (Oct. 2010). “Mining Your Own Evidence”. In: *Making Software*. O’Reilly Media, Inc. Chap. 27 (Cited on page 64).

Hofer, Birgit, Alexandre Perez, Rui Abreu, and Franz Wotawa (2015). “On the empirical evaluation of similarity coefficients for spreadsheets fault localization”. In: *Automated Software Engineering* 22.1, pp. 47–74 (Cited on page 11).

Hofer, Birgit and Franz Wotawa (2012). “Spectrum Enhanced Dynamic Slicing for better Fault Localization”. In: *ECAI 2012 - 20th European Conference on Artificial Intelligence. Including Prestigious Applications of Artificial Intelligence (PAIS-2012) System Demonstrations Track, Montpellier, France, August 27-31 , 2012*, pp. 420–425 (Cited on pages 10, 88).

Hogerle, Wolfgang, Friedrich Steimann, and Marcus Frenkel (2014). “More Debugging in Parallel”. In: *25th IEEE International Symposium on Software Reliability Engineering, ISSRE 2014, Naples, Italy, November 3-6, 2014*, pp. 133–143 (Cited on page 64).

Ishio, Takashi, Shogo Etsuda, and Katsuro Inoue (2012). “A lightweight visualization of interprocedural data-flow paths for source code reading”. In: *IEEE 20th International Conference on Program Comprehension, ICPC 2012, Passau, Germany, June 11-13, 2012*, pp. 37–46 (Cited on page 112).

Jones, James A. and Mary Jean Harrold (2005). “Empirical evaluation of the tarantula automatic fault-localization technique”. In: *20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005), November 7-11, 2005, Long Beach, CA, USA*, pp. 273–282 (Cited on pages 6, 7, 87).

Jones, James A., Mary Jean Harrold, and James F. Bowring (2007). “Debugging in Parallel”. In: *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2007, London, UK, July 9-12, 2007*, pp. 16–26 (Cited on page 64).

Jones, James A., Mary Jean Harrold, and John T. Stasko (2002). “Visualization of test information to assist fault localization”. In: *Proceedings of the 24th International Conference on Software Engineering, ICSE 2002, 19-25 May 2002, Orlando, Florida, USA*, pp. 467–477 (Cited on page 87).

Jost, Lou (2006). “Entropy and diversity”. In: *Oikos* 113.2, pp. 363–375 (Cited on page 27).

Jung, Ralf, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer (2018). “RustBelt: securing the foundations of the rust programming language”. In: *PACMPL* 2.POPL, 66:1–66:34 (Cited on page 119).

Just, René, Darioush Jalali, and Michael D. Ernst (2014a). “Defects4J: a database of existing faults to enable controlled testing studies for Java programs”. In: *International Symposium on Software Testing and Analysis, ISSTA ’14, San Jose, CA, USA - July 21 - 26, 2014*, pp. 437–440 (Cited on pages 36, 40).

Just, René, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser (2014b). “Are mutants a valid substitute for real faults in software testing?” In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, pp. 654–665 (Cited on page 45).

Kidwell, Peggy Aldrich (1998). “Stalking the Elusive Computer Bug”. In: *IEEE Annals of the History of Computing* 20.4, pp. 5–9 (Cited on page 1).

Kim, Sunghun, Thomas Zimmermann, E. James Whitehead Jr., and Andreas Zeller (2007). “Predicting Faults from Cached History”. In: *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*, pp. 489–498 (Cited on page 87).

de Kleer, Johan (1977). “Multiple Representations of Knowledge in a Mechanics Problem-Solver”. In: *Proceedings of the 5th International Joint Conference on Artificial Intelligence. Cambridge, MA, USA, August 22-25, 1977*, pp. 299–304 (Cited on pages 69, 70).

Ko, Andrew Jensen and Brad A. Myers (2008). “Debugging reinvented: asking and answering why and why not questions about program behavior”. In: *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*, pp. 301–310 (Cited on pages 2, 87).

– (2009). “Finding causes of program output with the Java Whyline”. In: *Proceedings of the 27th International Conference on Human Factors in Computing Systems, CHI 2009, Boston, MA, USA, April 4-9, 2009*, pp. 1569–1578 (Cited on page 87).

Ko, Andrew Jensen, Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung (2006). “An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks”. In: *IEEE Transactions on Software Engineering* 32.12, pp. 971–987 (Cited on page 113).

Korel, Bogdan and Janusz W. Laski (1988). “Dynamic Program Slicing”. In: *Information Processing Letters* 29.3, pp. 155–163 (Cited on page 88).

Kuhn, Rick, Raghu Kacker, Yu Lei, and Justin Hunter (2009). “Combinatorial Software Testing”. In: *IEEE Computer* 42.8, pp. 94–96 (Cited on page 119).

Kuhn, Rick, Dolores R. Wallace, and Albert M. Gallo (2004). “Software Fault Interactions and Implications for Software Testing”. In: *IEEE Trans. Software Eng.* 30.6, pp. 418–421 (Cited on page 119).

Kuipers, Benjamin (1986). “Qualitative Simulation”. In: *Artificial Intelligence* 29.3, pp. 289–338 (Cited on pages 69, 70).

Lange, Danny B. and Yuichi Nakamura (1997). “Object-Oriented Program Tracing and Visualization”. In: *IEEE Computer* 30.5, pp. 63–70 (Cited on page 92).

Lawrance, Joseph, Christopher Bogart, Margaret M. Burnett, Rachel K. E. Bellamy, Kyle Rector, and Scott D. Fleming (2013). “How Programmers Debug, Revisited: An Information Foraging Theory Perspective”. In: *IEEE Transactions on Software Engineering* 39.2, pp. 197–215 (Cited on pages 96, 113).

Lions, Jacques-Louis (1996). *Ariane 5: Flight 501 failure*. Tech. rep. ESA: Ariane 501 Inquiry Board (Cited on page 2).

Lucia, David Lo, Lingxiao Jiang, Ferdian Thung, and Aditya Budi (2014). “Extended comprehensive study of association measures for fault localization”. In: *Journal of Software: Evolution and Process* 26.2, pp. 172–219 (Cited on page 6).

Maletic, Jonathan I., Daniel J. Mosora, Christian D. Newman, Michael L. Collard, Andrew M. Sutton, and Brian P. Robinson (2011). “MosaiCode: Visualizing large scale software: A tool demonstration”. In: *Proceedings of the 6th IEEE International Workshop on Visualizing Software for Understanding and Analysis, VISSOFT 2011, Williamsburg, VA, USA, September 29-30, 2011*, pp. 1–4 (Cited on page 112).

Mann, Henry B. and Donald R. Whitney (1947). “On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other”. In: *The Annals of Mathematical Statistics* 18.1, pp. 50–60 (Cited on page 101).

Mao, Xiaoguang, Yan Lei, Ziying Dai, Yuhua Qi, and Chengsong Wang (2014). “Slice-based statistical fault localization”. In: *Journal of Systems and Software* 89, pp. 51–62 (Cited on page 88).

Mayer, Wolfgang and Markus Stumptner (2003). “Model-Based Debugging using Multiple Abstract Models”. In: *Proceedings of International Workshop on Automated and Analysis-Driven Debugging (AADEBUG’03)*, pp. 55–70 (Cited on page 3).

– (2008). “Evaluating Models for Model-Based Debugging”. In: *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), 15-19 September 2008, L’Aquila, Italy*, pp. 128–137 (Cited on pages 3, 119).

Miller, Joan C. and Clifford J. Maloney (1963). “Systematic mistake analysis of digital computer programs”. In: *Communications of the ACM* 6.2, pp. 58–63 (Cited on page 19).

Moreno, L., J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and K. Vijay-Shanker (2013). “Automatic generation of natural language summaries for Java classes”. In: *Proceedings of International Conference on Program Comprehension (ICPC’13)*, pp. 23–32 (Cited on page 120).

Moser, Raimund, Witold Pedrycz, and Giancarlo Succi (2008). “A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction”. In: *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*, pp. 181–190 (Cited on page 87).

Munaiah, Nuthan, Felivel Camilo, Wesley Wigham, Andrew Meneely, and Meiyappan Nagappan (2017). “Do bugs foreshadow vulnerabilities? An in-depth study of the chromium project”. In: *Empirical Software Engineering* 22.3, pp. 1305–1347 (Cited on page 64).

Nagappan, Nachiappan, Thomas Ball, and Andreas Zeller (2006). “Mining metrics to predict component failures”. In: *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*, pp. 452–461 (Cited on page 87).

Naish, Lee, Hua Jie Lee, and Kotagiri Ramamohanarao (2011). “A model for spectra-based software diagnosis”. In: *ACM Trans. Softw. Eng. Methodol.* 20.3, p. 11 (Cited on pages 6, 7, 47, 48).

Nguyen, Hoang Duong Thien, Dawei Qi, Abhik Roychoudhury, and Satish Chandra (2013). “SemFix: program repair via semantic analysis”. In: *35th International Conference on Software Engineering, ICSE ’13, San Francisco, CA, USA, May 18-26, 2013*, pp. 772–781 (Cited on pages 48, 58).

Nikolic, I., A. Kolluri, I. Sergey, P. Saxena, and A. Hobor (Feb. 2018). “Finding The Greedy, Prodigal, and Suicidal Contracts at Scale”. In: *ArXiv e-prints*. arXiv: `1802.06038 [cs.CR]` (Cited on page 3).

Pacheco, Carlos and Michael D. Ernst (2007). “Randoop: feedback-directed random testing for Java”. In: *Companion to the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, pp. 815–816 (Cited on page 43).

Parnin, Chris and Alessandro Orso (2011). “Are automated debugging techniques actually helping programmers?” In: *Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17-21, 2011*, pp. 199–209 (Cited on pages 10, 14, 58, 67).

Pauw, Wim De, Richard Helm, Doug Kimelman, and John M. Vlissides (1993). “Visualizing the Behavior of Object-Oriented Systems”. In: *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), Eighth Annual Conference, Washington, DC, USA, September 26 - October 1, 1993, Proceedings.* Pp. 326–337 (Cited on page 111).

Pauw, Wim De, David H. Lorenz, John M. Vlissides, and Mark N. Wegman (1998). “Execution Patterns in Object-Oriented Visualization”. In: *4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS), April 27-30, 1998, Eldorado Hotel, Santa Fe, New Mexico, USA*, p. 219 (Cited on pages 92, 111).

Pearson, Spencer, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, and Benjamin Keller (2017). “Evaluating and improving fault localization”. In: *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, pp. 609–620 (Cited on pages 10, 14, 67, 85).

Pelleg, Dan and Andrew W. Moore (2000). “X-means: Extending K-means with Efficient Estimation of the Number of Clusters”. In: *Proceedings of the Seventeenth International Conference on Machine Learning (ICML 2000), Stanford University, Stanford, CA, USA, June 29 - July 2, 2000*, pp. 727–734 (Cited on page 80).

Perez, Alexandre (2012). “Dynamic Code Coverage with Progressive Detail Levels”. Master’s thesis. Faculty of Engineering, University of Porto, Portugal (Cited on page 84).

Perez, Alexandre and Rui Abreu (2013). “Cues for scent intensification in debugging”. In: *IEEE 24th International Symposium on Software Reliability Engineering, ISSRE 2013, Pasadena, CA, USA, November 4-7, 2013 - Supplemental Proceedings*, pp. 120–125 (Cited on page 96).

– (2014). “A Diagnosis-based Approach to Software Comprehension”. In: *Proceedings of the 22nd International Conference on Program Comprehension, ICPC 2014, Hyderabad, India, June 2-3*, pp. 37–47 (Cited on pages 17, 91).

– (2016). “Framing Program Comprehension as Fault Localization”. In: *Journal of Software: Evolution and Process* 28.10, pp. 840–862 (Cited on pages 17, 91).

– (2018a). “Leveraging Qualitative Reasoning to Improve SFL”. In: *Proceedings of the 27th International Joint Conference on Artificial Intelligence and the 23rd European Conference on Artificial Intelligence, IJCAI-ECAI 2018, Stockholm, Sweden, July 13-19*, pp. 1–10 (Cited on pages 17, 67).

– (2018b). “Poster: A Qualitative Reasoning Approach to Spectrum-based Fault Localization”. In: *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 3*, pp. 1–2 (Cited on pages 17, 67).

Perez, Alexandre, Rui Abreu, and Marcelo d’Amorim (2017a). “Prevalence of Single-Fault Fixes and Its Impact on Fault Localization”. In: *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation, ICST’17, Tokyo, Japan, March 13-17*, pp. 12–22 (Cited on pages 17, 47).

Perez, Alexandre, Rui Abreu, and Arie van Deursen (2017b). “A Test-Suite Diagnosability Metric for Spectrum-based Fault Localization Approaches”. In: *Proceedings of the 39th International Conference on Software Engineering, ICSE’17, Buenos Aires, Argentina, May 20-28*, pp. 654–664 (Cited on pages 17, 19).

Perez, Alexandre, Rui Abreu, and André Riboira (2014). “A dynamic code coverage approach to maximize fault localization efficiency”. In: *Journal of Systems and Software* 90, pp. 18–28 (Cited on page 84).

Pinzger, Martin, Katja Grafenhain, Patrick Knab, and Harald C. Gall (2008). “A Tool for Visual Understanding of Source Code Dependencies”. In: *The 16th IEEE International Conference on Program Comprehension, ICPC 2008, Amsterdam, The Netherlands, June 10-13, 2008*, pp. 254–259 (Cited on page 112).

Piorkowski, David, Scott D. Fleming, Christopher Scaffidi, Christopher Bogart, Margaret M. Burnett, Bonnie E. John, Rachel K. E. Bellamy, and Calvin Swart (2012). “Reactive information foraging: an empirical investigation of theory-based recommender systems for programmers”. In: *CHI Conference on Human Factors in Computing Systems, CHI ’12, Austin, TX, USA - May 05 - 10, 2012*, pp. 1471–1480 (Cited on page 113).

Pirolli, Peter L. T. (2007). *Information Foraging Theory: Adaptive Interaction with Information*. 1st ed. Oxford University Press, Inc. (Cited on page 96).

Rahman, Foyzur, Christian Bird, and Premkumar T. Devanbu (2012). “Clones: what is that smell?” In: *Empirical Software Engineering* 17.4-5, pp. 503–530 (Cited on page 64).

Reiss, Steven P. (2003). “Visualizing Java in Action”. In: *Proceedings ACM 2003 Symposium on Software Visualization, San Diego, California, USA, June 11-13, 2003*, pp. 57–65, 210 (Cited on page 112).

Reiss, Steven P. and Manos Renieris (2001). “Encoding Program Executions”. In: *Proceedings of the 23rd International Conference on Software Engineering, ICSE 2001, 12-19 May 2001, Toronto, Ontario, Canada*, pp. 221–230 (Cited on pages 92, 111).

Renieris, Manos and Steven P. Reiss (1999). “Almost: Exploring program traces”. In: *Proceedings of Workshop on New Paradigms in Information Visualization and Manipulation (NPIVM’99)*, pp. 70–77 (Cited on page 92).

Richardson, Debra J. and Margaret C. Thompson (1993). “An Analysis of Test Data Selection Criteria Using the RELAY Model of Fault Detection”. In: *IEEE Transactions on Software Engineering* 19.6, pp. 533–553 (Cited on page 15).

Robles, Gregorio and Jesús M. González-Barahona (2005). “Developer identification methods for integrated data from various sources”. In: *ACM SIGSOFT Software Engineering Notes* 30.4, pp. 1–5 (Cited on page 64).

Rogerson, Simon (2002). “The chinook helicopter disaster”. In: *The Institute for the Management of Information Systems (IMIS)* 12.2 (Cited on page 2).

Romero, Pablo, Benedict du Boulay, Richard Cox, Rudi Lutz, and Sallyann Bryant (2007). “Debugging strategies and tactics in a multi-representation software environment”. In: *International Journal of Man-Machine Studies* 65.12, pp. 992–1009 (Cited on page 113).

Schmidt, D. C. (2006). “Model-Driven Engineering”. In: *IEEE Computer* 39.2, pp. 25–31 (Cited on page 119).

Schuler, David and Andreas Zeller (2011). “Assessing Oracle Quality with Checked Coverage”. In: *Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST 2011, Berlin, Germany, March 21-25, 2011*, pp. 90–99 (Cited on page 45).

Shamshiri, Sina, René Just, José Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri (2015). “Do Automatically Generated Unit Tests Find Real Faults? An Empirical Study of Effectiveness and Challenges (T)”. In: *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, pp. 201–211 (Cited on page 85).

Shannon, Claude E. (2001). “A mathematical theory of communication”. In: *Mobile Computing and Communications Review* 5.1, pp. 3–55 (Cited on page 24).

Shapiro, Samuel S. and Martin B. Wilk (1965). “An Analysis of Variance Test for Normality (Complete Samples)”. In: *Biometrika* 52.3-4, pp. 591–611 (Cited on pages 42, 61, 83, 101).

Sliwerski, Jacek, Thomas Zimmermann, and Andreas Zeller (2005). “When do changes induce fixes?” In: *Proceedings of the 2005 International Workshop on Mining Software Repositories, MSR 2005, Saint Louis, Missouri, USA, May 17, 2005* (Cited on pages 48, 64).

Sohn, Jeongju and Shin Yoo (2017). “FLUCCS: using code and change metrics to improve fault localization”. In: *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*, pp. 273–283 (Cited on pages 64, 88).

de Souza, Higor Amario and Marcos Lordello Chaim (2013). “Adding context to fault localization with integration coverage”. In: *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, pp. 628–633 (Cited on page 87).

Spool, Jared M., Christine Perfetti, and David Brittan (2004). *Designing for the scent of information*. User Interface Engineering (Cited on page 113).

Steimann, Friedrich and Marcus Frenkel (2012). “Improving Coverage-Based Localization of Multiple Faults Using Algorithms from Integer Linear Programming”. In: *23rd IEEE International Symposium on Software Reliability Engineering, ISSRE 2012, Dallas, TX, USA, November 27-30, 2012*, pp. 121–130 (Cited on page 64).

Steimann, Friedrich, Marcus Frenkel, and Rui Abreu (2013). “Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators”. In: *International Symposium on Software Testing and Analysis, ISSTA ’13, Lugano, Switzerland, July 15-20, 2013*, pp. 314–324 (Cited on pages 10, 14, 55, 67).

Stengel, Michael, Mathias Frisch, Sven Apel, Janet Feigenspan, Christian Kästner, and Raimund Dachselt (2011). “View infinity: a zoomable interface for feature-oriented software development”. In: *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, pp. 1031–1033 (Cited on page 112).

Stephenson, Arthur G., Daniel R. Mulville, Frank H. Bauer, Greg A. Dukeman, Peter Norvig, Lia S. LaPiana, Peter J. Rutledge, David Folta, and Robert Sackheim (1999). “Mars Climate Orbiter Mishap Investigation Board Phase I Report”. In: *NASA, Washington, DC* (Cited on page 2).

Tassey, Gregory (2002). *The Economic Impacts of Inadequate Infrastructure for Software Testing* (Cited on pages 1, 2).

Tiarks, Rebecca (2011). “What Programmers Really Do - An Observational Study”. In: *Softwaretechnik-Trends* 31.2 (Cited on pages 16, 92).

Trümper, Jonas, Jürgen Döllner, and Alexandru Telea (2013). “Multiscale visual comparison of execution traces”. In: *IEEE 21st International Conference on Program Comprehension, ICPC 2013, San Francisco, CA, USA, 20-21 May, 2013*, pp. 53–62 (Cited on page 112).

Wang, Shaowei and David Lo (2014). “Version history, similar report, and structure: putting them together for improved bug localization”. In: *22nd International Conference on Program Comprehension, ICPC 2014, Hyderabad, India, June 2-3, 2014*, pp. 53–63 (Cited on page 87).

Wang, Xinming, Shing-Chi Cheung, Wing Kwong Chan, and Zhenyu Zhang (2009). “Taming coincidental correctness: Coverage refinement with context patterns to improve fault localization”. In: *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*, pp. 45–55 (Cited on pages 15, 45).

Wilcoxon, Frank (1945). “Individual Comparisons by Ranking Methods”. In: *Biometrics Bulletin* 1.6, pp. 80–83 (Cited on pages 42, 61, 83).

Wilde, Norman and Michael C. Scully (1995). “Software reconnaissance: Mapping program features to code”. In: *Journal of Software Maintenance* 7.1, pp. 49–62 (Cited on page 112).

Williams, Brian C. and Johan de Kleer (1991). “Qualitative Reasoning about Physical Systems: A Return to Roots”. In: *Artificial Intelligence* 51.1-3, pp. 1–9 (Cited on pages 68, 69).

Wong, W. Eric, Vidroha Debroy, Ruizhi Gao, and Yihao Li (2014). “The DStar Method for Effective Software Fault Localization”. In: *IEEE Transactions on Reliability* 63.1, pp. 290–308 (Cited on pages 6, 47).

Wong, W. Eric, Vidroha Debroy, Yihao Li, and Ruizhi Gao (2012). “Software Fault Localization Using DStar (D*)”. In: *Sixth International Conference on Software Security and Reliability, SERE 2012, Gaithersburg, Maryland, USA, 20-22 June 2012*, pp. 21–30 (Cited on page 6).

Wong, W. Eric, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa (2016). “A survey of software fault localization”. In: *IEEE Transactions on Software Engineering* (Cited on pages 6, 10).

Wotawa, Franz (2010). “Fault Localization Based on Dynamic Slicing and Hitting-Set Computation”. In: *Proceedings of the 10th International Conference on Quality Software, QSIC 2010, Zhangjiajie, China, 14-15 July 2010*, pp. 161–170 (Cited on page 88).

Xie, Xiaoyuan, Tsong Yueh Chen, Fei-Ching Kuo, and Baowen Xu (2013). “A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization”. In: *ACM Transactions on Software Engineering and Methodology* 22.4, 31:1–31:40 (Cited on page 15).

Xu, Xiaofeng, Vidroha Debroy, W. Eric Wong, and Donghui Guo (2011). “Ties within Fault Localization rankings: Exposing and Addressing the Problem”. In: *International Journal of Software Engineering and Knowledge Engineering* 21.6, pp. 803–827 (Cited on page 15).

Xuan, Jifeng and Martin Monperrus (2014). “Test case purification for improving fault localization”. In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, pp. 52–63 (Cited on page 45).

Yazdanshenas, Amir Reza and Leon Moonen (2012). “Tracking and visualizing information flow in component-based systems”. In: *IEEE 20th International Conference on Program Comprehension, ICPC 2012, Passau, Germany, June 11-13, 2012*, pp. 143–152 (Cited on page 112).

Ye, Lina and Philippe Dague (2010). “Diagnosability Analysis of Discrete Event Systems with Autonomous Components”. In: *ECAI 2010 - 19th European Conference on Artificial Intelligence, Lisbon, Portugal, August 16-20, 2010, Proceedings*, pp. 105–110 (Cited on page 119).

Yoo, Shin and Mark Harman (2010). “Using hybrid algorithm for Pareto efficient multi-objective test suite minimisation”. In: *Journal of Systems and Software* 83.4, pp. 689–701 (Cited on page 45).

Yoo, Shin, Xiaoyuan Xie, Fei-Ching Kuo, Tsong Yueh Chen, and Mark Harman (2014). “No Pot of Gold at the End of Program Spectrum Rainbow: Greatest Risk Evaluation Formula Does Not Exist”. In: *RN/14/14*. University College London. (Cited on page 6).

Yu, Yanbing, James A. Jones, and Mary Jean Harrold (2008). “An empirical study of the effects of test-suite reduction on fault localization”. In: *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*, pp. 201–210 (Cited on page 45).

Zaidman, Andy (2006). “Scalability Solutions for Program Comprehension through Dynamic Analysis”. In: *10th European Conference on Software Maintenance and Reengineering (CSMR 2006), 22-24 March 2006, Bari, Italy*, pp. 327–330 (Cited on page 92).

Zeller, Andreas (1999). “Yesterday, My Program Worked. Today, It Does Not. Why?” In: *Software Engineering - ESEC/FSE’99, 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Toulouse, France, September 1999, Proceedings*, pp. 253–267 (Cited on page 120).

Zhang, Feng, Audris Mockus, Iman Keivanloo, and Ying Zou (2016). “Towards building a universal defect prediction model with rank transformed predictors”. In: *Empirical Software Engineering* 21.5, pp. 2107–2145 (Cited on pages 64, 87).

Zhang, Yonggang, Juergen Rilling, and Volker Haarslev (2006). “An Ontology-Based Approach to Software Comprehension - Reasoning about Security Concerns”. In: *30th Annual International Computer Software and Applications Conference, COMPSAC 2006, Chicago, Illinois, USA, September 17-21, 2006. Volume 1*, pp. 333–342 (Cited on page 103).

Zhao, Wei, Lu Zhang, Yin Liu, Jiasu Sun, and Fuqing Yang (2006). “SNIAFL: Towards a Static Noninteractive Approach to Feature Location”. In: *ACM Transactions on Software Engineering and Methodology* 15.2, pp. 195–226 (Cited on page 113).

Zhu, Hong, Patrick A. V. Hall, and John H. R. May (1997). “Software Unit Test Coverage and Adequacy”. In: *ACM Computing Surveys* 29.4, pp. 366–427 (Cited on page 19).