

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO



FEUP

Dynamic Code Coverage with Progressive Detail Levels

Alexandre Campos Perez

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Rui Maranhão (PhD)

Co-Supervisor: André Riboira (MSc)

18th June, 2012

Dynamic Code Coverage with Progressive Detail Levels

Alexandre Campos Perez

Mestrado Integrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

Chair: Ademar Manuel Teixeira de Aguiar (PhD)

External Examiner: João Alexandre Baptista Vieira Saraiva (PhD)

Supervisor: Rui Filipe Maranhão de Abreu (PhD)

18th June, 2012

This work is financed by the ERDF – European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by National Funds through the FCT - Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project PTDC/EIA-CCO/116796/2010.



Resumo

Hoje em dia, a localização de componentes de software responsáveis por uma avaria é uma das tarefas mais dispendiosas e propensas a erros no processo de desenvolvimento de software.

Para melhorar a eficiência do processo de depuração, algum esforço já foi feito para automaticamente auxiliar a deteção e localização de falhas de software. Isto levou à criação de ferramentas de depuração estatística como Tarantula, Zoltar e GZoltar. Estas ferramentas utilizam informação recolhida a partir de dados de cobertura de código e do resultado das execuções dos casos de teste para retornar uma lista dos locais mais prováveis de conter uma falha.

Estas ferramentas de localização de falhas, apesar de úteis, têm alguns problemas de dimensionamento devido à necessidade de analisar dados de cobertura com granularidade fina. O *overhead* de instrumentação, que em alguns casos pode ser tão elevado como 50%, é a principal causa para a sua ineficiência.

Esta tese propõe uma nova abordagem para este problema, evitando tanto quanto possível o elevado nível de detalhe de cobertura, mas continuando a utilizar as técnicas comprovadas que as ferramentas de localização de falhas empregam.

Esta abordagem, chamada Dynamic Code Coverage (DCC), consiste na utilização de uma instrumentação inicial mais grosseira, obtendo dados de cobertura apenas para componentes de grandes dimensões (*p.e.*, classes). Em seguida, o detalhe da instrumentação de certos componentes é progressivamente aumentado, com base nos resultados intermédios fornecidos pelas mesmas técnicas de localização de falhas utilizadas em ferramentas atuais.

Para avaliar a eficácia da abordagem proposta, foi realizada uma avaliação empírica, injetando falhas em quatro projetos de software. A avaliação empírica demonstra que a abordagem DCC reduz o *overhead* de execução que existe nas técnicas atuais de localização de falhas, e também apresenta ao utilizador um relatório de diagnóstico de falha mais conciso. Foi observada uma redução do tempo de execução de 27% em média e uma redução do tamanho do relatório de diagnóstico de 63% em média.

Abstract

Nowadays, locating software components responsible for observed failures is one of the most expensive and error-prone tasks in the software development process.

To improve the debugging process efficiency, some effort was already made to automatically assist the detection and location of software faults. This led to the creation of statistical debugging tools such as Tarantula, Zoltar and GZoltar. These tools use information gathered from code coverage data and the result of test executions to return a list of potential faulty locations.

Although helpful, fault localization tools have some scaling problems because of the fine-grained coverage data they need to perform the fault localization analysis. Instrumentation overhead, which in some cases can be as high as 50% is the main cause for their inefficiency.

This thesis proposes a new approach to this problem, avoiding as much as possible the high level of coverage detail, while still using the proven techniques these fault localization tools employ.

This approach, named Dynamic Code Coverage (**DCC**), consists of using a coarser initial instrumentation, obtaining only coverage traces for large components. Then, the instrumentation detail of certain components is progressively increased, based on the intermediate results provided by the same techniques employed in current fault localization tools.

To assess the validity of our proposed approach, an empirical evaluation was performed, injecting faults in four real-world software projects. The empirical evaluation demonstrates that the **DCC** approach reduces the execution overhead that exists in spectrum-based fault localization, and even presents a more concise potential fault ranking to the user. We have observed execution time reductions of 27% on average and diagnostic report size reductions of 63% on average.

Acknowledgements

This thesis project would certainly not have been the same without the help of several people and organizations. I would like to take a moment to acknowledge and thank them.

First, I would like to thank Faculdade de Engenharia da Universidade do Porto for providing me with the knowledge that I have gained these last few years. I would also like to express my utmost gratitude to my supervisors, Prof. Dr. Rui Maranhão and André Riboira for their support, motivation and insight. Their guidance and feedback helped me greatly throughout this project and I could not imagine having better mentors and advisors for my MSc thesis. It was a pleasure working with them and I hope to be able to work with them again in the future.

A special thanks goes to João Santos, José Carlos de Campos, Nuno Cardoso and Francisco Silva for the all the laughs, support and insightful suggestions during my research at the Software Engineering Laboratory.

I would also like to give my sincere thanks to my all friends and family for being so supportive and understanding of my absence during stressful periods.

Last, but certainly not least, I would like to thank my parents, Jesus and Maria Filomena, for their unending support throughout my life.

Porto, 18th June, 2012

Alexandre Perez

Contents

1	Introduction	1
1.1	Context	2
1.2	Concepts and Definitions	3
1.3	Motivation	4
1.4	Research Question	5
1.5	A Dynamic Code Coverage Approach	5
1.6	Document Structure	6
2	State of the art	9
2.1	Traditional Debugging	9
2.1.1	Assertions	9
2.1.2	Breakpoints	9
2.1.3	Profiling	10
2.1.4	Code Coverage	10
2.2	Statistical Debugging	12
2.2.1	Tarantula	14
2.2.2	Zoltar	15
2.2.3	EzUnit	16
2.2.4	GZoltar	18
2.3	Reasoning Approaches	18
2.3.1	Model-Based Diagnosis	18
2.3.2	Model-Based Software Debugging	19
2.4	Discussion	19
3	Dynamic Code Coverage	21
3.1	Motivational Example	21
3.2	Dynamic Code Coverage Algorithm	22
3.3	Discussion	26
4	Tooling	29
4.1	GZoltar Toolset	29
4.2	Modifications and Improvements	31
4.3	Dynamic Code Coverage Prototype	32
4.4	Discussion	33
5	Empirical Evaluation	35
5.1	Experimental Setup	35
5.2	Experimental Results	36

CONTENTS

5.3	Threats to Validity	41
6	Conclusions and Future Work	43
6.1	State of the art of Debugging Tools	43
6.2	Proposed Solution	43
6.3	Main Contributions	44
6.4	Publications	44
6.5	Future Work	45
	References	47
A	Publications	51
A.1	Fault Localization using Dynamic Code Coverage	52
A.2	GZoltar: an Eclipse plug-in for Testing and Debugging	53
A.3	A Dynamic Code Coverage Approach to Maximize Fault Localization Efficiency	61

List of Figures

1.1	Software development process.	1
1.2	First actual case of bug being found.	2
1.3	Progressive detail of a component.	6
2.1	Instrumentation Code Insertion [TH02].	11
2.2	Input to Spectrum-based Fault Localization (SFL) [JAvG09a].	12
2.3	Tarantula interface.	14
2.4	Zoltar interface [JAvG09a].	15
2.5	SFL's similarity coefficients performance comparison [Abr09].	16
2.6	EzUnit interface.	17
2.7	EzUnit call graph.	17
2.8	GZoltar interface [Rib11].	18
3.1	SFL output example.	21
3.2	Component filters.	24
3.3	DCC output example.	25
	(a) First iteration	25
	(b) Second iteration	25
	(c) Third iteration	25
4.1	GZoltar's visualizations: Sunburst and Treemap.	30
4.2	RZoltar's interface.	30
4.3	Statement failure probability markers.	31
5.1	NanoXML time execution results.	37
	(a) Coefficient filter	37
	(b) Percentage filter	37
5.2	org.jacoco.report time execution results.	38
	(a) Coefficient filter	38
	(b) Percentage filter	38
5.3	XML-Security time execution results.	39
	(a) Coefficient filter	39
	(b) Percentage filter	39
5.4	JMeter time execution results.	40
	(a) Coefficient filter	40
	(b) Percentage filter	40

LIST OF FIGURES

List of Tables

2.1	Code Coverage tools comparison.	11
2.2	Example of SFL technique with Ochiai coefficient.	13
5.1	Experimental Subjects.	36

LIST OF TABLES

List of Algorithms

1	Dynamic Code Coverage.	23
---	--------------------------------	----

LIST OF ALGORITHMS

Abbreviations

DCC Dynamic Code Coverage

IDE Integrated Development Environment

JVM Java Virtual Machine

LOC Line Of Code

MBD Model-Based Diagnosis

MBSD Model-Based Software Debugging

SDT Statistical Debugging Tool

SFL Spectrum-based Fault Localization

SUT System Under Test

Chapter 1

Introduction

The software development process generally follows four main phases: a requirements and design phase, after that an implementation phase, followed by a testing phase, and finally the release. All these steps can be defined, and estimated, with a high degree of certainty.

However, in most, if not all, software projects, some tests fail. Because of that, cycles are introduced in the process, in which another task has to be performed – the so-called debugging phase (see Figure 1.1).

Most of the time, the debugging phase consists of changing the implementation so that faults are eliminated. However, in some cases, the system design itself can be at fault, and has to be modified. In this thesis, we will focus on the software implementation debugging.

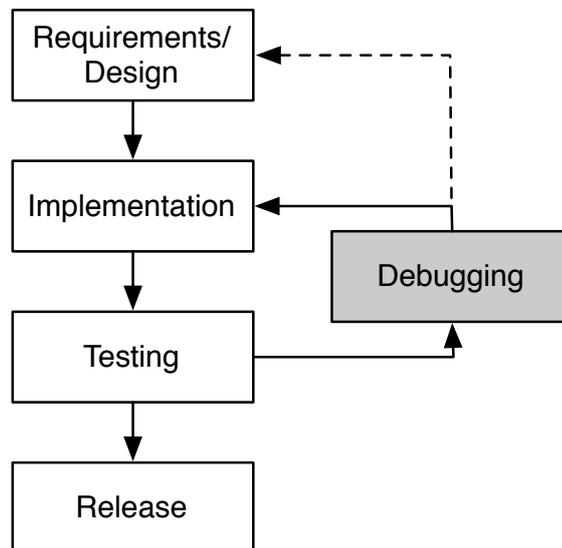


Figure 1.1: Software development process.

The debugging phase consumes a huge amount of a project's resources [Tas02, HS02]. Furthermore, one cannot estimate with a high degree of certainty the cost of this phase (in terms of

time and also money). For this reason, it is important to minimize the impact that debugging has in the development process.

Software debugging tools and methodologies almost always existed, but they were fairly ineffective and *ad-hoc*. Currently, there are some tools that automate this process by returning the most likely locations of containing a fault. However, these tools do not scale because they need to instrument code at a fine-grained detail level.

This thesis' main goal is to improve the efficiency of automated debugging techniques so that developers spend less time locating faults and thus minimizing the cost of this cumbersome phase.

1.1 Context

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 1.2) [Kid98]. This incident coined the use of the terms “bug”, “debug” and “debugging” in the field of computer science.



Figure 1.2: First actual case of bug being found.

Introduction

In software development, a large amount of resources is spent in the debugging phase. It is estimated that testing and debugging activities can easily range from 50 to 75 percent of the total development cost [HS02]. This is due to the fact that the process of detecting, locating and fixing faults in the source code, is not trivial and rather 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 [KM08].

If this debugging task is not thoroughly conducted, even bigger costs may arise. In fact, a landmark study performed in 2002 indicated that software defects constitute an annual \$60 billion cost to the US economy alone [Tas02].

The first debugging techniques to be used consisted of utilizing prints and stack traces, assertions, breakpoints, coverage information and profiling. These techniques, which sometimes are called traditional debugging techniques, are quite ineffective by themselves and rather *ad-hoc* in nature.

Traditional debugging techniques have several limitations. They rely heavily on the developer's intuition and suffer from a considerable execution overhead, since many combinations of program executions have to be examined. Furthermore, in most cases, developers only use failing test cases to diagnose a certain defect, disregarding useful information about passing test cases. Traditional debugging also requires developers to have a comprehensive knowledge of the program under test.

In order to improve the debugging efficiency, this process needs to be automated. Some effort was already made to automatically assist the detecting and locating steps of the software debugging phase. This led to the creation of automatic fault localization tools, namely Zoltar [JAvG09a] and Tarantula [JHS02]. The tools instrument the source code to obtain code coverage traces for each execution (also known as program spectra), which are then analyzed to return a list of potential faulty locations.

To improve the exploration and intuitiveness of that potential faulty locations list, an Eclipse¹ plugin was also developed – GZoltar [RA10]. This tool provides fault localization functionality to an Integrated Development Environment (IDE), with several interactive visualization options, as well as testing selection and prioritization functionalities.

1.2 Concepts and Definitions

Throughout this thesis, we use the following terminology [ALRL04]:

- 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.

¹Eclipse integrated development environment – <http://www.eclipse.org/>

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.

Definition 1 *A software program Π is formed by a sequence of one or more M statements.*

Given its dynamic nature, central to the fault localization technique considered in this thesis is the existence of a test suite.

Definition 2 *A test suite $T = \{t_1, \dots, t_N\}$ 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 $|T| = N$.*

Definition 3 *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.*

1.3 Motivation

It is essential to find ways to minimize the software testing and debugging impact on a project's resources. However, it is imperative that the software quality (*i.e.*, correctness) is not compromised. While some defects can be tolerated by users (or even not perceived at all), others may cause severe financial and/or life-threatening consequences. Examples of well-known 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 [Lio96, Dow97];
- The crash of a British Royal Air Force Chinook due to a software defect in the helicopter's engine control computer, killing 29 people [Rog02].

For this reason, automatic debugging tools are essential to aid developers in maintaining their software project's quality. Nowadays, automatic fault localization techniques can aid developers/testers in pinpointing the root cause of software failures, and thereby reducing the debugging effort. Amongst the most diagnostic-effective techniques is Spectrum-based Fault Localization (SFL).

SFL is a statistical technique that uses abstraction of program traces (also known as program spectra) to correlate software component (e.g., statements, methods, classes) activity with program failures [AZGV09, LFY⁺06, WWQZ08]. As SFL is typically used to aid developers in identifying what is the root cause of observed failures, it is used with high level of granularity (*i.e.*, statement level).

Statistical approaches to debugging are very attractive because of the relatively small overhead with respect to CPU time and memory requirement [AZGV09, AZvG09]. However, gathering the

input information, per test case, to yield the diagnostic ranking may still impose a considerable (CPU time) overhead. This is particularly the case for resource constrained environments, such as embedded systems.

As said before, typically, [SFL](#) is used at development-time at a statement level granularity (since debugging requires to locate the faulty statement). But not all components need to be inspected at such fine grain granularity. In fact, components that are unlikely to be faulty may not need to be inspected. This way, by removing instrumentation from unlikely locations, more projects will be suitable to be debugged with these fault localization techniques, since the imposed overhead will be reduced.

1.4 Research Question

The main research question that we are trying to answer with this work is the following:

- How can a fault localization approach that instruments less software components obtain similar diagnostic results when compared with [SFL](#), while reducing execution overhead?

The main objective of this thesis is to devise a fault localization approach that is comparable to [SFL](#) in terms of diagnostic results, while trying to be more lightweight regarding the instrumentation used to obtain the characterization of the program executions (*i.e.*, the program spectra).

This way, bigger projects will be able to use these debugging methodologies than before (especially resource constrained projects). It is also expected that due to this instrumentation decrease, the fault localization process will be shorter in terms of execution time.

1.5 A Dynamic Code Coverage Approach

This thesis proposes a technique, coined Dynamic Code Coverage ([DCC](#)).

This technique automatically adjusts the granularity per software component. First, our approach instruments the source code using a coarse granularity (e.g., package level in Java) and the fault localization is executed. Then, it is decided which components are *expanded* based on the output of the fault localization technique. With expanding we mean changing the granularity of the instrumentation to the next detail level (e.g., in Java, for instance, instrument classes, then methods, and finally statements), behaving like the diagram in [Figure 1.3](#). This expansion can be done in different ways, either selecting the components whose fault coefficient is above a certain threshold, or selecting the first ranked components, according to a set percentage.

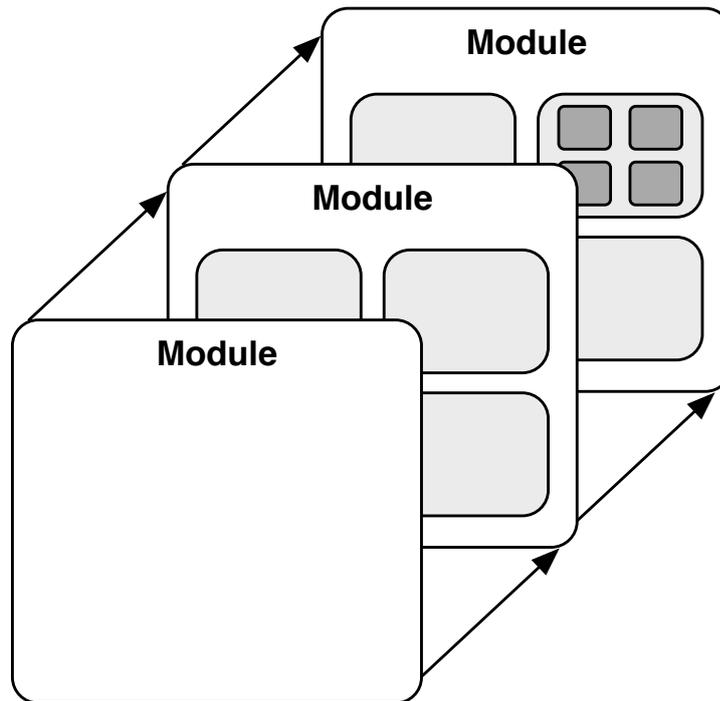


Figure 1.3: Progressive detail of a component.

1.6 Document Structure

This document will be structured as follows:

Chapter 2 contains the state of the art in this project’s field. Traditional and statistical debugging techniques and tools, and model-based reasoning approaches to debugging are presented in this chapter.

Chapter 3 details the DCC algorithm. A motivational example is presented to show the inefficiencies of current approaches, followed by the DCC algorithm description and its main advantages and shortcomings.

Chapter 4 presents the tool chosen to host the DCC prototype – GZoltar – as well as some modifications that have been made to this tool.

Chapter 5 presents our empirical evaluation setup and findings.

Chapter 6 presents some conclusions about the work and describes the main contributions of this thesis. After that, some future work challenges are presented.

Lastly, Appendix A contains the accepted scientific publications during this work, submitted to *IJUP’12*² and *ASE’12*³. A publication currently being prepared for submission into *ICST’13*⁴

²The 5th Meeting of Young Researchers of University of Porto, 2012

³The 27th IEEE/ACM International Conference on Automated Software Engineering, 2012

⁴The 6th IEEE International Conference on Software Testing, Verification and Validation, 2013

Introduction

is also included.

Note that most of the work presented in this thesis has been submitted for publication at *ICST'13*.

Introduction

Chapter 2

State of the art

In software development, a large amount of resources is spent in debugging [Tas02, HS02]. This process consists of three major phases:

- **Detecting** a fault in a program's behavior.
- **Locating** said fault in the source code.
- **Fixing** the code to eliminate the fault.

This is not trivial and rather error-prone. For this reason, many tools were created to assist developers in this process, each tool having its advantages and its disadvantages. This chapter presents the state of the art of debugging techniques.

2.1 Traditional Debugging

In this section, some traditional debugging techniques and tools will be described, namely assertions, breakpoints, profiling and code coverage.

2.1.1 Assertions

Assertions are formal constraints that the developer can use to specify *what* the system is supposed to do (rather than *how*) [Ros95].

These constructs are generally predefined macros that expand into an *if* statement that aborts the execution if the expression inside the assertion evaluates to false.

Assertions can be seen, then, as permanent defense mechanisms for runtime fault detection.

2.1.2 Breakpoints

A breakpoint specifies that the control of a program execution should transfer to the user when a specified instruction is reached [CLR05]. The execution is stopped and the user can inspect and manipulate its state (*e.g.* the user can read and change variable values). It is also possible to

perform a step-by-step execution after the breakpoint. This is particularly useful to observe a bug as it develops, and to trace it to its origin.

There are other types of breakpoints, namely data breakpoints and conditional breakpoints. Data breakpoints (also called watchpoints [SPS06]) transfer control to the user when the value of an expression changes. This expression may be a value of a variable, or multiple variables combined by operators (*e.g.*: $a + b$). Conditional breakpoints only stop the execution if a certain user-specified predicate is true, thus reducing the frequency of user-application interaction.

Breakpoints are available in most modern Integrated Development Environments (IDEs).

2.1.3 Profiling

Profiling is a dynamic analysis that gathers some metrics from the execution of a program, such as memory usage and frequency and duration of function calls. Profiling's main use is to aid program optimization, but it is also useful for debugging purposes, such as:

- Knowing if functions are being called more or less often than expected;
- Finding if certain portions of code execute slower than expected or if they contain memory leaks;
- Investigating the behavior of lazy evaluation strategies.

Known profiling tools include GNU's gprof [GKM82] and the Eclipse plugin TPTP¹.

2.1.4 Code Coverage

Code coverage is an analysis method that determines which parts of the System Under Test (SUT) have been executed (covered) during a system test run [GvVEB06].

Using code coverage in conjunction with tests, it is possible to see which Lines Of Code (LOCs), methods or classes were covered in a specific test (depending on the set level of detail). With this information, it is possible to identify which components were involved in a system failure, narrowing the search for the faulty component that made the test fail.

Table 2.1 presents a non-exhaustive list of code coverage tools available in the market, with the languages they support and the detail levels they provide (adapted from [YLW06]). It is worth to note that we have limited the scope of our research to code coverage tools for imperative and object oriented languages. This way, a comparison using coverage measurement detail levels can be established.

¹Eclipse Test & Performance Tools Platform Project – <http://www.eclipse.org/tptp/>

State of the art

	Website	Language(s)	Line	Decision	Method	Class
Agitar	http://www.agitar.com/	Java	✓	✓	✓	✓
Bullseye	http://www.bullseye.com/	C/C++		✓	✓	
Clover	http://www.atlassian.com/software/clover/	Java, .NET	✓	✓	✓	
Cobertura	http://cobertura.sourceforge.net/	Java	✓	✓		
Dynamic	http://www.dynamic-memory.com/	C/C++	✓	✓	✓	
EclEmma	http://www.eclEmma.org/	Java	✓	✓	✓	✓
gcov	http://gcc.gnu.org/onlinedocs/gcc/Gcov.html	C/C++	✓			
Insure++	http://www.parasoft.com/jsp/products/insure.jsp	C/C++	✓			
JCover	http://www.mmsindia.com/JCover.html	Java	✓	✓	✓	✓
JTest	http://www.parasoft.com/jsp/products/jtest.jsp	Java, .NET	✓	✓		
PurifyPlus	http://www.ibm.com/software/awdtools/purifyplus/	C/C++, Java, .NET	✓		✓	
SD	http://www.semdesigns.com/	C/C++, Java, C#, PHP, COBOL	✓	✓	✓	✓
TCAT	http://www.soft.com/Products/Coverage/	C/C++, Java	✓	✓	✓	✓

Table 2.1: Code Coverage tools comparison.

In order to obtain information about what components were covered in each run, these Code Coverage tools have to instrument the system code. This instrumentation will monitor each component and register if they were executed.

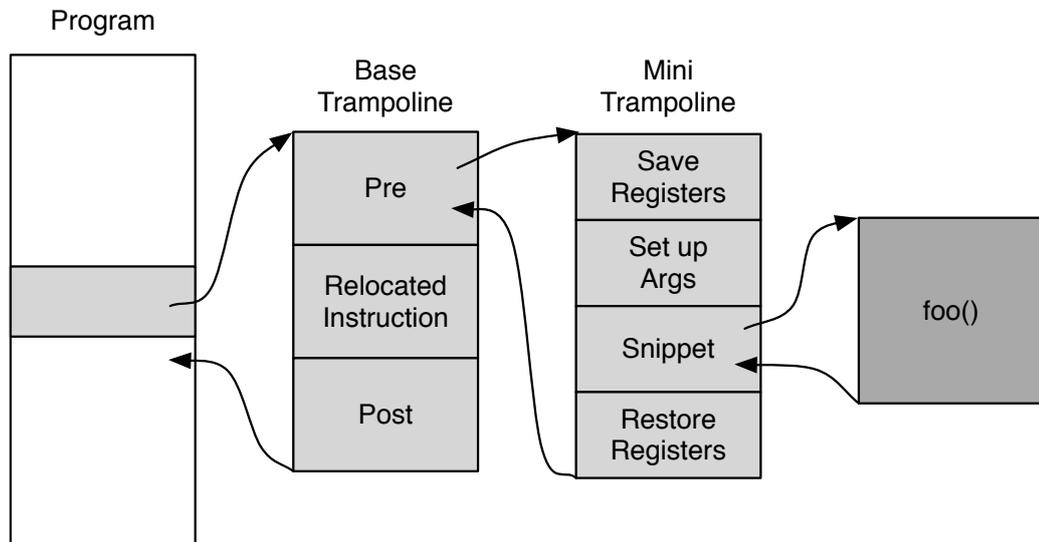


Figure 2.1: Instrumentation Code Insertion [TH02].

Instrumentation code, as seen in Figure 2.1, relies in a series of trampolines to a desired function `foo()` before the desired instructions. In the case of Code Coverage tools, `foo()` will

register that the instruction was touched by the execution.

This instrumentation introduces overhead to the system’s execution during the testing and debugging phases. According to Yang, et al [YLW06], Code Coverage tools, which have to instrument at a LOC level, introduce execution speed overheads of up to 50%.

2.2 Statistical Debugging

Statistical Debugging Tools (SDTs) use statistical techniques to calculate the probability of a certain software component of the SUT containing faults. The most effective statistical technique is Spectrum-based Fault Localization (SFL) [Abr09].

SFL exploits information from passed and failed system runs. A passed run is a program execution that is completed correctly, and a failed run is an execution where an error was detected [JAvG09a]. The criteria for determining if a run has passed or failed can be from a variety of different sources, namely test case results and program assertions, among others. The information gathered from these runs is their code coverage (also called program spectra [AZG06]).

A program spectra is a characterization of a program’s execution on a dataset [RBDL97]. This collection of data, gathered at runtime, provides a view on the dynamic behavior of a program. The data consists of counters or flags for each software component. Different program spectra exist [HRS⁺00], such as path-hit spectra, data-dependence-hit spectra, and block-hit spectra.

As explained in section 2.1.4, in order to obtain information about which components were covered in each execution, the program’s source code needs to be instrumented, similarly to code coverage tools [YLW06]. This instrumentation will monitor each component and register those that were executed. Components can be of several detail granularities, such as classes, methods, and lines of code.

The hit spectra of N runs constitutes a binary $N \times M$ matrix A , where M corresponds to the instrumented components of the program. Information of passed and failed runs is gathered in a N -length vector e , called the error vector. The pair (A, e) serves as input for the SFL technique, as seen in Figure 2.2.

$$\begin{array}{c}
 N \text{ spectra} \\
 \left[\begin{array}{cccc}
 a_{11} & a_{12} & \cdots & a_{1M} \\
 a_{21} & a_{22} & \cdots & a_{2M} \\
 \vdots & \vdots & \ddots & \vdots \\
 a_{N1} & a_{N2} & \cdots & a_{NM}
 \end{array} \right]
 \end{array}
 \begin{array}{c}
 M \text{ components} \\
 \\
 \end{array}
 \begin{array}{c}
 \text{error} \\
 \text{detection} \\
 \left[\begin{array}{c}
 e_1 \\
 e_2 \\
 \vdots \\
 e_N
 \end{array} \right]
 \end{array}$$

Figure 2.2: Input to SFL [JAvG09a].

With this input, fault localization consists in identifying what columns of the matrix A resemble the vector e the most. For that, several different similarity coefficients can be used [JD88].

State of the art

mid() { int x, y, z, m;	Runs						Coef.
	1	2	3	4	5	6	
1: read("Enter 3 numbers:", x, y, z);	●	●	●	●	●	●	0.41
2: m = z;	●	●	●	●	●	●	0.41
3: if (y<z) {	●	●	●	●	●	●	0.41
4: if (x<y)	●	●			●	●	0.50
5: m = y;		●					0.0
6: else if (x<z)	●				●	●	0.58
7: m = y; //BUG	●				●		0.71
8: } else {			●	●			0.0
9: if (x>y)			●	●			0.0
10: m = y;			●				0.0
11: else if (x>z)				●			0.0
12: m = x;							0.0
13: }							0.0
14: print("Middle number is:", m);	●	●	●	●	●	●	0.41
}							
Pass/fail status:	✓	✓	✓	✓	✗	✓	

Table 2.2: Example of SFL technique with Ochiai coefficient.

One of them is the Ochiai coefficient [AZvG07], used in the molecular biology domain. Ochiai is defined as follows:

$$s_o(j) = \frac{n_{11}(j)}{\sqrt{(n_{11}(j) + n_{01}(j)) \times (n_{11}(j) + n_{10}(j))}} \quad (2.1)$$

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

$$n_{pq}(j) = |\{i \mid a_{ij} = p \wedge e_i = q\}| \quad (2.2)$$

In Table 2.2 it is shown an example of the SFL technique, using the Ochiai coefficient (adapted from [JH05]). To improve this example's legibility, the coverage matrix and the error detection vector were transposed. In this example, the SUT is a function named `mid()` that reads three integer numbers and prints the median value. This program contains a fault on line 7 – it should read `m = x;`.

Six test cases were run, and their coverage information for each LOC can be seen to the right. At the bottom there is also the pass/fail status for each run – which corresponds to the error detection vector e . Then, the similarity coefficient was calculated for each line using the Ochiai coefficient (2.1). These results represent the likelihood of a certain line containing a fault. The bigger the coefficient, the more likely it is of a line containing a fault. Therefore, these coefficients

can be ranked to form an ordered list of the probable faulty locations.

In this specific example, the highest coefficient is in line 7 – the faulty **LOC**. The **SFL** technique has successfully performed the fault localization.

2.2.1 Tarantula

Tarantula² [JHS02] is a visual debugging system that is used to debug projects written in the C language. This tool is being developed at Georgia Tech and uses **SFL** for the fault localization.

Tarantula runs test suites against the **SUT** and displays the calculated probability of each **LOC** by highlighting them accordingly – varying from red (maximum failure probability) to green (minimum failure probability). In Figure 2.3 it is shown the tool's interface. Tarantula has the ability to analyze the whole system at once, which is convenient for debugging large projects, as well as gathering management metrics about the project.

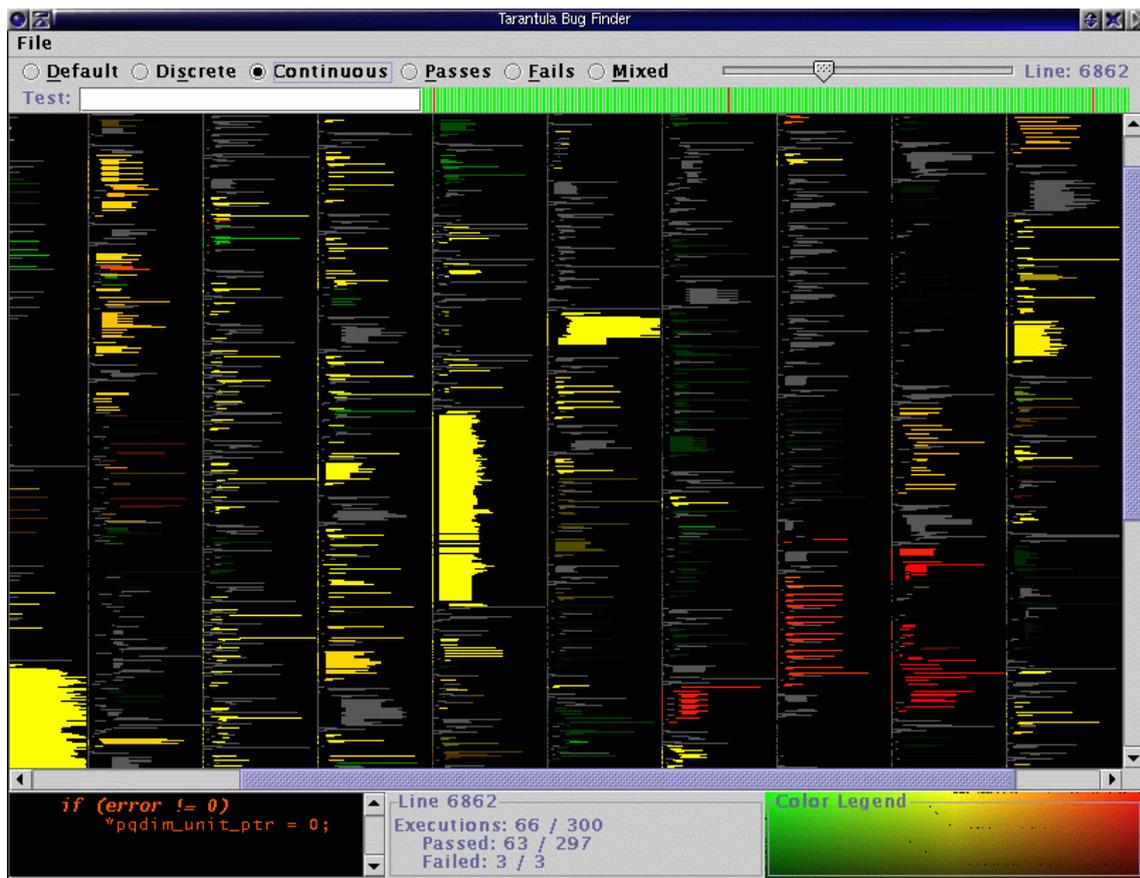


Figure 2.3: Tarantula interface.

²Tarantula - Fault Localization via Visualization – <http://pleuma.cc.gatech.edu/aristotle/Tools/tarantula/>

For the similarity coefficient, Tarantula uses the following coefficient [AZvG07, JH05]:

$$s_T(j) = \frac{\frac{n_{11}(j)}{n_{11}(j)+n_{01}(j)}}{\frac{n_{11}(j)}{n_{11}(j)+n_{01}(j)} + \frac{n_{10}(j)}{n_{10}(j)+n_{00}(j)}} \quad (2.3)$$

However, studies [AZvG07] have shown that this coefficient under-performs Ochiai (2.1). Furthermore, Tarantula is currently not available for download.

2.2.2 Zoltar

Zoltar [JAvG09a] is a tool that implements fault localization in C/C++ projects (see Figure 2.4). This tool currently presents superior performance compared with similar tools, and has implemented several algorithms, namely Barinel, Ochiai and Tarantula [Abr09]. Figure 2.5 shows the comparison between several Zoltar algorithms.

Zoltar was developed at Delft University of Technology (TUDelft), and was the base of Rui Abreu's PhD thesis [Abr09]. This tool had substantial academic recognition, and won the *Best Demo Award* prize at *The 24th IEEE/ACM International Conference on Automated Software Engineering (ASE'09)* with the publication *Zoltar: A Toolset for Automatic Fault Localization* [JAvG09b].

Zoltar has already proven to be effective in real world uses, from the development of embedded devices, such as TV sets, to sizable internal software projects.

```

Software Analyzer v0.2.4
Delft University of Technology

Operating Mode
Runs
-Spectra
Invariants
Exit

SFL analysis
-----

spectrum name          Basic_Blocks
number of components   35
SFL coefficient         Ochiai

rank—score—component info
0 0.577350 textVal.c:43 -
1 0.500000 textVal.c:29 -
2 0.447214 textVal.c:17 -
3 0.447214 textVal.c:19 -
4 0.447214 textVal.c:19 -
5 0.447214 textVal.c:20 -
6 0.447214 textVal.c:22 -
7 0.447214 textVal.c:26 -
8 0.447214 textVal.c:28 -
9 0.447214 textVal.c:33 -

pgup, pgdn ^v backspace

```

Figure 2.4: Zoltar interface [JAvG09a].

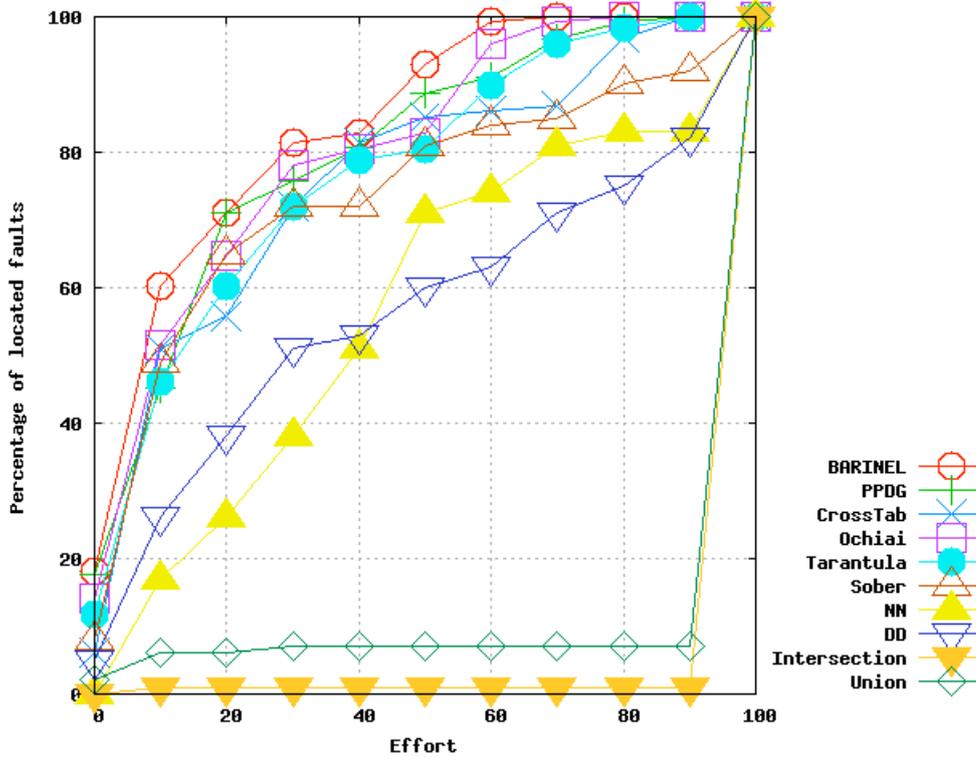


Figure 2.5: SFL's similarity coefficients performance comparison [Abr09].

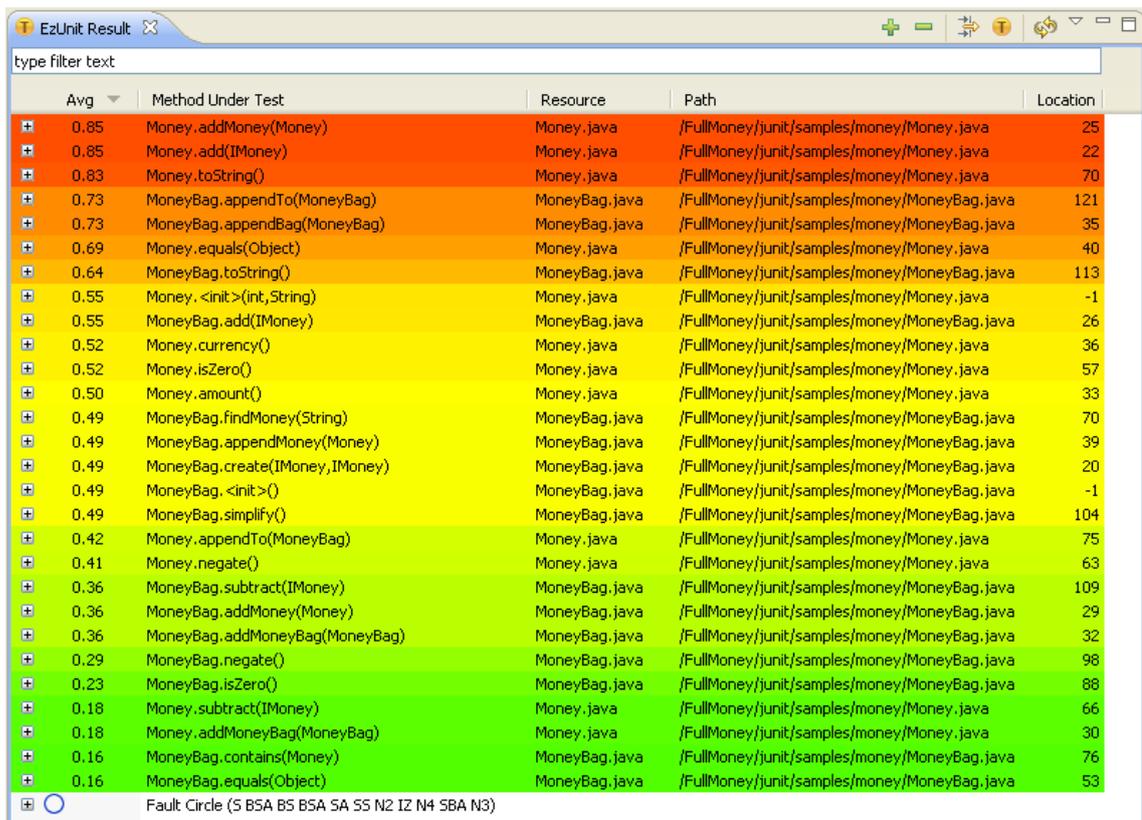
2.2.3 EzUnit

EzUnit³ is a statistical debugging tool under development at University of Hagen. This tool is integrated into the Eclipse IDE as a plugin, and is used to debug Java projects that use JUnit test cases.

After performing the fault localization, EzUnit displays a list of the code blocks ranked by their failure probability in a separate view in the Eclipse IDE (see Figure 2.6). Each line of the list is highlighted with a color representing the failure probability, ranging from red (corresponding to the code blocks that are most likely to contain a failure) to green (least likely). EzUnit also provides a call-graph view of a certain test (see Figure 2.7). Each node in the graph corresponds to a code block, and has the same coloring scheme as the failure probability list view.

³EzUnit – Easing the Debugging of Unit Test Failures – <http://www.fernuni-hagen.de/ps/prjs/EzUnit4/>

State of the art



The screenshot shows the EzUnit Result window with a table of test results. The table has columns for Avg, Method Under Test, Resource, Path, and Location. The results are sorted by Avg, with the highest value at the top. The methods listed include Money.addMoney(Money), Money.add(IMoney), Money.toString(), MoneyBag.appendTo(MoneyBag), MoneyBag.appendBag(MoneyBag), Money.equals(Object), MoneyBag.toString(), Money.<init>(int,String), MoneyBag.add(IMoney), Money.currency(), Money.isZero(), Money.amount(), MoneyBag.findMoney(String), MoneyBag.appendMoney(Money), MoneyBag.create(IMoney,IMoney), MoneyBag.<init>(), MoneyBag.simplify(), Money.appendTo(MoneyBag), Money.negate(), MoneyBag.subtract(IMoney), MoneyBag.addMoney(Money), MoneyBag.addMoneyBag(MoneyBag), MoneyBag.negate(), MoneyBag.isZero(), Money.subtract(IMoney), Money.addMoneyBag(MoneyBag), MoneyBag.contains(Money), and MoneyBag.equals(Object). The Location column shows line numbers, with some values being -1.

Avg	Method Under Test	Resource	Path	Location
0.85	Money.addMoney(Money)	Money.java	/FullMoney/junit/samples/money/Money.java	25
0.85	Money.add(IMoney)	Money.java	/FullMoney/junit/samples/money/Money.java	22
0.83	Money.toString()	Money.java	/FullMoney/junit/samples/money/Money.java	70
0.73	MoneyBag.appendTo(MoneyBag)	MoneyBag.java	/FullMoney/junit/samples/money/MoneyBag.java	121
0.73	MoneyBag.appendBag(MoneyBag)	MoneyBag.java	/FullMoney/junit/samples/money/MoneyBag.java	35
0.69	Money.equals(Object)	Money.java	/FullMoney/junit/samples/money/Money.java	40
0.64	MoneyBag.toString()	MoneyBag.java	/FullMoney/junit/samples/money/MoneyBag.java	113
0.55	Money.<init>(int,String)	Money.java	/FullMoney/junit/samples/money/Money.java	-1
0.55	MoneyBag.add(IMoney)	MoneyBag.java	/FullMoney/junit/samples/money/MoneyBag.java	26
0.52	Money.currency()	Money.java	/FullMoney/junit/samples/money/Money.java	36
0.52	Money.isZero()	Money.java	/FullMoney/junit/samples/money/Money.java	57
0.50	Money.amount()	Money.java	/FullMoney/junit/samples/money/Money.java	33
0.49	MoneyBag.findMoney(String)	MoneyBag.java	/FullMoney/junit/samples/money/MoneyBag.java	70
0.49	MoneyBag.appendMoney(Money)	MoneyBag.java	/FullMoney/junit/samples/money/MoneyBag.java	39
0.49	MoneyBag.create(IMoney,IMoney)	MoneyBag.java	/FullMoney/junit/samples/money/MoneyBag.java	20
0.49	MoneyBag.<init>()	MoneyBag.java	/FullMoney/junit/samples/money/MoneyBag.java	-1
0.49	MoneyBag.simplify()	MoneyBag.java	/FullMoney/junit/samples/money/MoneyBag.java	104
0.42	Money.appendTo(MoneyBag)	Money.java	/FullMoney/junit/samples/money/Money.java	75
0.41	Money.negate()	Money.java	/FullMoney/junit/samples/money/Money.java	63
0.36	MoneyBag.subtract(IMoney)	MoneyBag.java	/FullMoney/junit/samples/money/MoneyBag.java	109
0.36	MoneyBag.addMoney(Money)	MoneyBag.java	/FullMoney/junit/samples/money/MoneyBag.java	29
0.36	MoneyBag.addMoneyBag(MoneyBag)	MoneyBag.java	/FullMoney/junit/samples/money/MoneyBag.java	32
0.29	MoneyBag.negate()	MoneyBag.java	/FullMoney/junit/samples/money/MoneyBag.java	98
0.23	MoneyBag.isZero()	MoneyBag.java	/FullMoney/junit/samples/money/MoneyBag.java	88
0.18	Money.subtract(IMoney)	Money.java	/FullMoney/junit/samples/money/Money.java	66
0.18	Money.addMoneyBag(MoneyBag)	Money.java	/FullMoney/junit/samples/money/Money.java	30
0.16	MoneyBag.contains(Money)	MoneyBag.java	/FullMoney/junit/samples/money/MoneyBag.java	76
0.16	MoneyBag.equals(Object)	MoneyBag.java	/FullMoney/junit/samples/money/MoneyBag.java	53

Fault Circle (5 B5A B5 B5A SA S5 N2 IZ N4 SBA N3)

Figure 2.6: EzUnit interface.

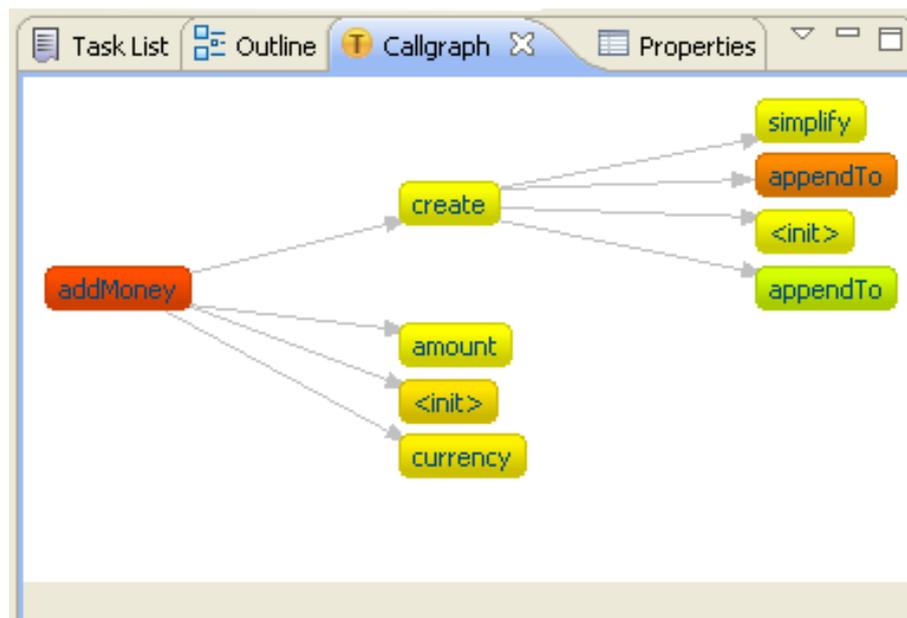


Figure 2.7: EzUnit call graph.

2.2.4 GZoltar

GZoltar⁴ [RA10, RAR11] is a framework for automating testing and debugging projects written in Java. It is an Eclipse-based, Java implementation of Zoltar that integrates with frameworks such as JUnit. It also provides powerful hierarchical visualization and interaction options to developers (such as a sunburst view, see Figure 2.8).

The GZoltar tool was the base of André Ribeiro's MSc thesis [Rib11] and is under active development at Faculdade de Engenharia da Universidade do Porto. The work detailed in this thesis is aimed at improving this tool. Further information about the GZoltar project is available in Section 4.1.

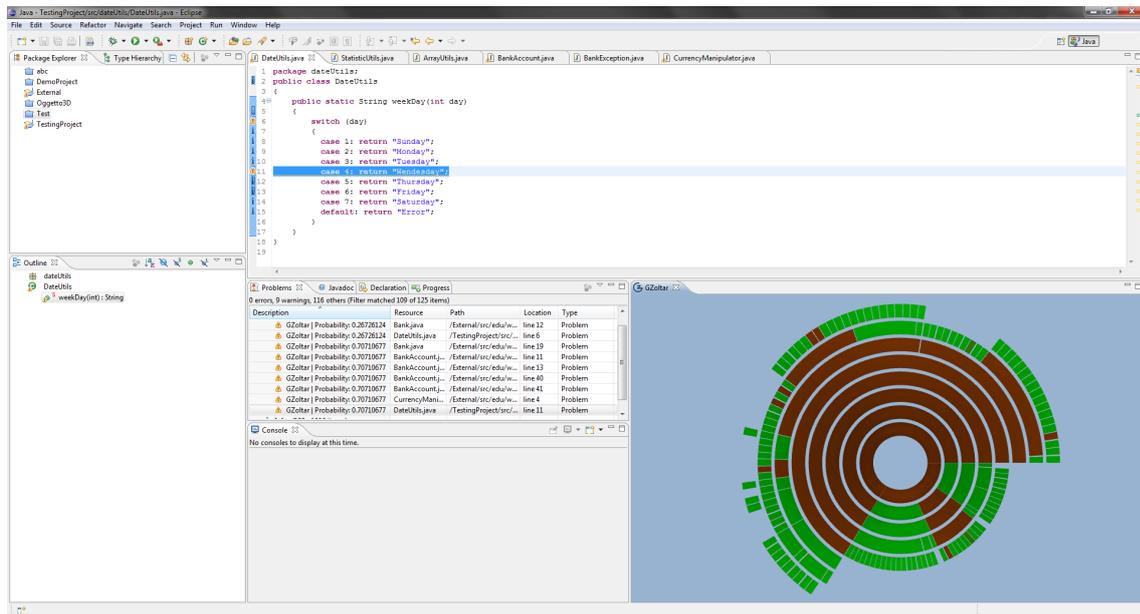


Figure 2.8: GZoltar interface [Rib11].

2.3 Reasoning Approaches

Reasoning approaches to fault localization use prior knowledge of the system, such as required component behavior and interconnection, to build a model of the system behavior. An example of a reasoning technique is Model-Based Diagnosis (MBD) (see, e.g., [dKW87]).

2.3.1 Model-Based Diagnosis

In MBD, a diagnosis is obtained by logical inference from the *static* model of the system, combined with a set of run-time observations. Traditional MBD systems require the model to be supplied by the system's designer, whereas the description of the observed behavior is gathered through direct measurements. The difference between the behavior described by the model and the

⁴The GZoltar Project – <http://www.gzoltar.com/>

observed behavior can then be used to identify components that may explain possible deviations from normal behavior [MS07].

In practice, as a formal description of the program is required, the task of using MBD can be difficult. This is particularly due to (1) the large scope of current software projects, where formal models are rarely made available, and (2) the maintenance problems that arise throughout development, since changes in functionality are likely to happen. Furthermore, formal models usually do not describe a system's complete behavior, being restricted to a particular component of the system.

2.3.2 Model-Based Software Debugging

In order to address some of the issues that traditional MBD has, Model-Based Software Debugging (MBSD) exchanges the roles of the model and the observations. In this technique, instead of requiring the designer to formally specify the intended behavior, a model is automatically inferred from the actual program. This means that the model reflects all the faults that exist in the program. The correct behavior specification in this technique is described in the system's test cases, which specify the expected output for a certain input.

Well-known approaches to MBSD include the approaches of Friedrich, Stumptner, and Wotawa [FSW99, FSW96], Nica and Wotawa [NW08], Wotawa, Stumptner, and Mayer [WSM02], and Mayer and Stumptner [MS07]. However, MBSD has problems concerning scalability – the computational effort required to create a model of a large program forbids its use in real-life applications [MS08].

2.4 Discussion

Currently, the most effective debugging tools are the SDTs. These tools, with minimal effort from the user, return a ranked list of potential faulty locations. By comparison, traditional debugging tools require more user effort for locating faults, and are fairly *ad-hoc*.

SDTs not only require less effort, but they also help improve the debugging process by automating it. This is particularly useful for regression testing.

However, SDTs have some flaws regarding performance, as the SUT has to be instrumented. Studies have shown that instrumentation can hit execution time by as much as 50% [YLW06], so this approach of fault localization is particularly inefficient for large, real systems, that contain hundreds of thousands of LOCs. Also, while parallelization can help minimize the impact of instrumentation, we may not be able to use it in every situation (and particularly while dealing with resource constrained projects).

Other debugging techniques, use reasoning approaches to debugging, such as MBSD. However, the computational effort required to create a model of a large application is very high.

State of the art

Chapter 3

Dynamic Code Coverage

In this chapter, we present a motivational example showing why traditional [SFL](#) approaches can be inefficient, mainly due to the overhead caused by instrumenting every line of code. Afterwards, we propose and algorithm, coined Dynamic Code Coverage ([DCC](#)), that mitigates those inefficiencies by gradually adjusting the instrumentation granularity of each software component of the [SUT](#).

3.1 Motivational Example

Suppose a program responsible for controlling a television set is being debugged. Consider that such program has three main high-level modules:

1. Audio and video processing;
2. Teletext decoding and navigation;
3. Remote-control input.

If one is to use [SFL](#) to pinpoint the root cause of observed failures, hit spectra for the entire application have to be gathered. Furthermore, the hit spectra have to be of a fine granularity, such as [LOC](#) level, so that the fault is more easily located.

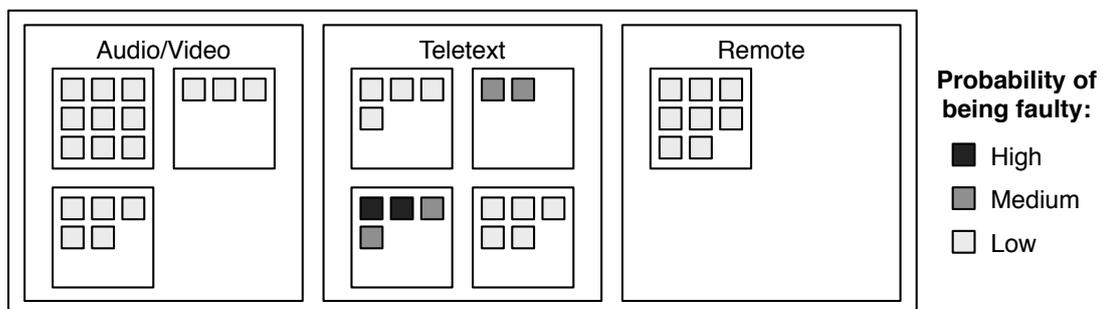


Figure 3.1: [SFL](#) output example.

An output of the **SFL** technique applied to this specific example can be seen in Figure 3.1. The smaller squares represent each **LOC** of the program, which are grouped into methods, and then into the three main modules of the program under test.

As seen in Figure 3.1, every **LOC** in the program has an associated fault coefficient that represents the probability of that component being faulty. In this example, the bottom-left function of the teletext decoding and navigation module has two **LOCs** with high probability of being faulty, and other two with medium probability. The upper-right function of the teletext module also contains two medium probability **LOCs**.

There are, however, many **LOCs** with low probability of containing a fault. In fact, in some methods, and even entire modules, such as the audio/video processing module and the remote-control module, all components have low probability of being at fault. Such low probability is an indication that the fault might be located elsewhere, and thus these components need not to be inspected first.

As **SFL** needs to have information about the entire program spectra to perform an analysis on the most probable fault locations, this can lead to scalability problems, as every **LOC** has to be instrumented. As previously stated in Section 2.1.4, instrumentation can hit execution time by as much as 50% in code coverage tools that use similar instrumentation techniques [YLW06]. As such, fault localization that uses hit spectra is acceptable for debugging software applications, but may be impractical for large, real-world, and resource-constrained projects that contain hundreds of thousands of **LOCs**.

In order to make **SFL** amenable to large, real, and resource-constrained applications, a way to avoid instrumenting the entire program must be devised, while still having a fine granularity for the most probable locations in the fault localization results.

3.2 Dynamic Code Coverage Algorithm

In order to solve the scaling problems that automated fault localization tools have, it is proposed a **DCC** approach. This method uses, at first, a coarser granularity level of instrumentation and progressively increases the instrumentation detail of potential faulty components.

DCC is shown in Algorithm 1. It takes as parameters *System*, *TestSuite*, *InitialGranularity* and *FinalGranularity*. These parameters correspond to the **SUT**, its test case set, and the initial and final instrumentation detail levels, respectively.

Algorithm 1 Dynamic Code Coverage.

```

1: procedure DCC(System, TestSuite, InitialGranularity, FinalGranularity)
2:    $\mathcal{R} \leftarrow \emptyset$ 
3:    $\mathcal{F} \leftarrow \textit{System}$ 
4:    $\mathcal{T} \leftarrow \textit{TestSuite}$ 
5:    $\mathcal{G} \leftarrow \textit{InitialGranularity}$ 
6:   repeat
7:     INSTRUMENT( $\mathcal{F}$ ,  $\mathcal{G}$ )
8:      $(A, e) \leftarrow \text{RUNTESTS}(\mathcal{T})$ 
9:      $\mathcal{C} \leftarrow \text{SFL}(A, e)$ 
10:     $\mathcal{F} \leftarrow \text{FILTER}(\mathcal{C})$ 
11:     $\mathcal{R} \leftarrow \text{UPDATE}(\mathcal{R}, \mathcal{F})$ 
12:     $\mathcal{T} \leftarrow \text{NEXTTESTS}(\textit{TestSuite}, A, \mathcal{F})$ 
13:     $\mathcal{G} \leftarrow \text{NEXTGRANULARITY}(\mathcal{F})$ 
14:  until ISFINALGRANULARITY( $\mathcal{F}$ , FinalGranularity)
15:  return  $\mathcal{R}$ 
16: end procedure
    
```

First, an empty report \mathcal{R} is created. After that, a list of the components to instrument \mathcal{F} is initialized with all *System* components. Similarly, the list of test cases to run in each iteration \mathcal{T} is initialized with all test cases in *TestSuite*. An initial granularity \mathcal{G} is also initialized with the desired initial exploration granularity *InitialGranularity*, which can be set from a class level to a [LOC](#) level.

After the initial assignments, the algorithm will start its iteration phase in line 6. At the start of each iteration, every component in the list \mathcal{F} is instrumented with the granularity \mathcal{G} with the method INSTRUMENT. What this method does is to alter these components so that their execution is registered in the program spectra.

Afterwards, the test cases \mathcal{T} are run with the method RUNTESTS. Its output is a hit spectra matrix A for all the previously instrumented components, and the error vector e , that states what tests passed and what tests failed. As explained in Section 2.2, these are the necessary inputs for spectrum-based fault localization, performed in line 9. This SFL method calculates, for each instrumented component, its failure coefficient using the Ochiai coefficient, previously shown in equation 2.1.

Following the fault localization step, the components are passed through a FILTER that eliminates the low probability ones according to a set threshold, and the list \mathcal{F} is updated, as well as the fault localization report \mathcal{R} .

In line 12, the test case set is updated to run only the tests that touch the current components \mathcal{F} . Such tests can be retrieved by analyzing the coverage matrix A .

The last step in the iteration is to update the instrumentation granularity for next iterations. Method NEXTGRANULARITY finds the coarser granularity in all the components of list \mathcal{F} , and updates that granularity to the next level of detail.

Every iteration is tested for recursion with ISFINALGRANULARITY, that returns true if every component in the list \mathcal{F} is at the desired final granularity defined in *FinalGranularity*. This final

granularity can be of different detail levels, such as method level or **LOC** level, according to the needs of the software project being tested. If the `ISFINALGRANULARITY` condition is not met, a new iteration is performed.

Lastly, the **DCC** algorithm returns the fault localization report \mathcal{R} . \mathcal{R} contains diagnosis candidates of different granularity, typically with the top ones at the statement-level granularity.

Coefficient Filter (> 0.6)					
0.9	0.8	0.8	0.7	0.5	0.1
0.7	0.2	0.2	0.1	0.0	0.0

Percentage Filter (50%)					
0.9	0.8	0.8	0.7	0.5	0.1
0.7	0.2	0.2	0.1	0.0	0.0

Figure 3.2: Component filters.

DCC's performance is very dependent on the `FILTER` function, which is responsible to decide whether or not it is required to zoom-in¹ in a given component. Although many filters may be plugged into the algorithm, in this thesis we study the impact of two filters (see Figure 3.2 for an illustration):

- Coefficient filter C_f – components above the **SFL** coefficient threshold C_f are expanded.
- Percentage filter P_f – the first $P_f\%$ components are expanded.

To illustrate the overhead reduction, let us revisit the motivational example given in Section 3.1. If use the **DCC** approach to debug this program, we get the output of each iteration of the algorithm as shown in Figure 3.3. In this example, a filter responsible for not exploring components with low probability of containing faults is being used. In particular, the algorithm executes as follows:

1. The three modules – Audio/Video, Teletext, and Remote – are instrumented at the module level. Upon running the tests and **SFL**, the only component with high probability of being faulty is the Teletext module. See Figure 3.3a.

¹In this context, zooming-in is to explore the inner components of a given component.

Dynamic Code Coverage

2. The Teletext module is instrumented at a method level. After that, the tests that touch the Teletext module are run. Fault localization states that the upper-right (UR) and the bottom-left (BL) functions have medium and high probability of being faulty, respectively. See Figure 3.3b.
3. The UR and BL functions are instrumented at the **LOC** level. After the tests that touch those functions are run and fault localization is performed, every **LOC** in those functions has an associated fault coefficient. As all the non-filtered components are of **LOC** granularity, the execution is terminated. See Figure 3.3c.

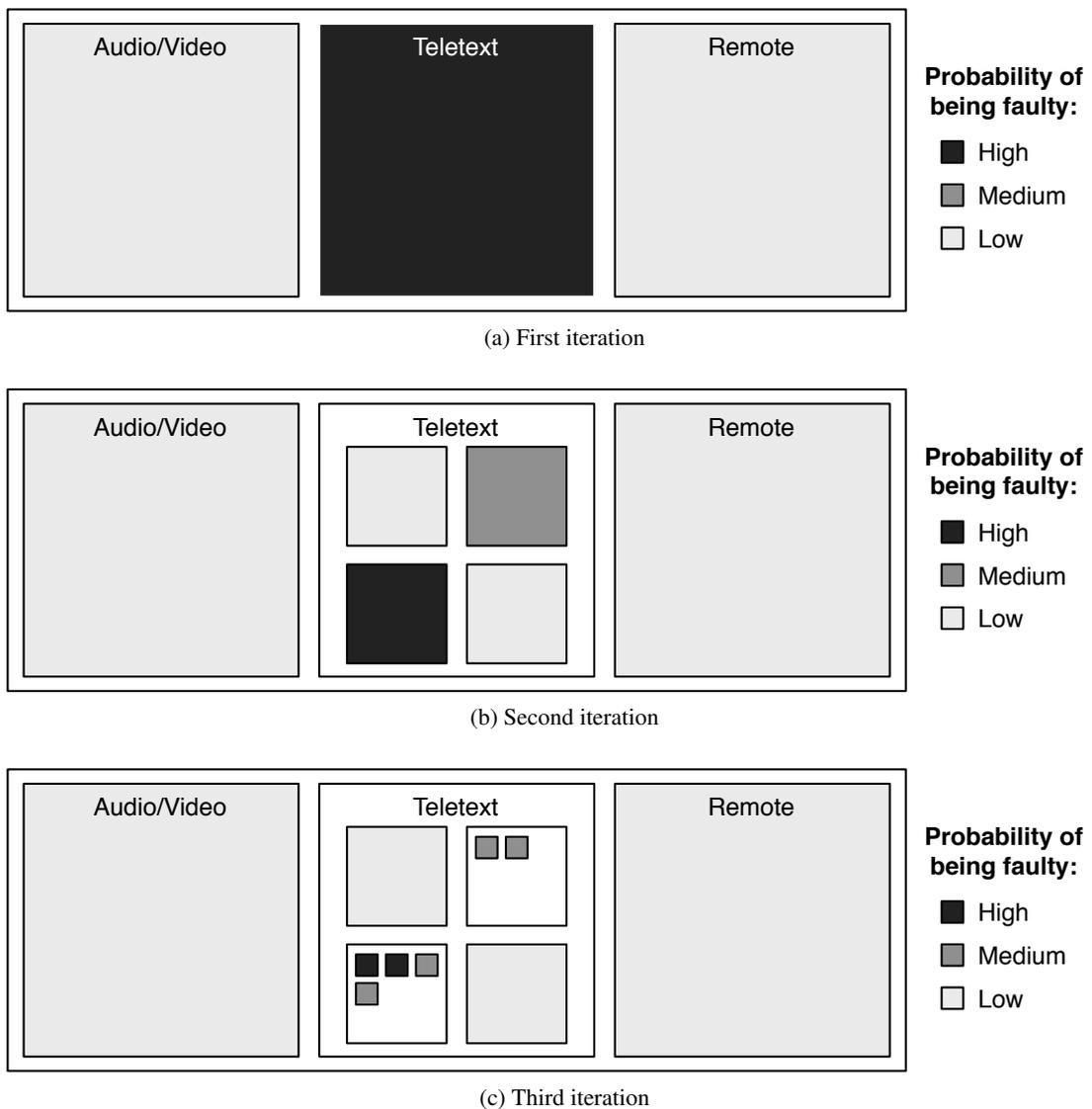


Figure 3.3: DCC output example.

This approach, besides only reporting LOCs which are more likely to contain a fault, also needs to instrument less software components – 13 in total. Compared to the pure **SFL** approach

of Section 3.1, where 40 components were instrumented, DCC has reduced instrumentation (thus, its overhead) by 67.5%, while producing the same good results.

3.3 Discussion

DCC is a fault localization algorithm that instruments software components at a low level of detail, and progressively increases the instrumentation detail of software components likely to be at fault.

The main advantages of our DCC algorithm are twofold.

1. The instrumentation overhead in the program execution decreases. This is due to the fact that not every LOC is instrumented – only the LOCs most likely to contain a fault will be instrumented at that level of detail.
2. In every iteration, the generated program spectra matrices, seen in line 8 of Algorithm 1, will be shorter in size when compared to traditional SFL. That way, the fault coefficient calculation, described in Section 2.2, will be inherently faster, as there are fewer components to calculate.

The iterative nature of the DCC algorithm also provides some benefits. In each iteration, the algorithm is walking towards a solution, narrowing down the list of components which are likely to contain a fault. As such some information about those components can be made available, directing the developer to the fault location even before the algorithm is finished. Also, as low probability components are being filtered, the final report will be shorter, providing the developer with a more concise fault localization report.

The ability that DCC has to stop at any desired granularity level of detail can also provide benefits. For instance, some defects may be successfully diagnosed upon inspecting the list of faulty methods in a project, so an instrumentation at a LOC may not be always necessary. A more important use case of this cutoff ability, though, is to combine complementary fault localization methods. For example, DCC can be used to obtain the top ranked methods in a program, and then employ the MBSD technique, detailed in Section 2.3.2, only on these software components. This way, we are obtaining more accurate fault hypotheses due to the use of MBSD, with significantly less computation required.

However, this algorithm still poses a couple of shortcomings. As DCC relies on program hierarchy to devise different detail levels of instrumentation, some projects may not be suited to be debugged with this technique. In fact, while object-oriented programming languages, which have at least three clearly defined detail levels (*i.e.*, class, method and LOC), perform well with DCC, other programming languages with different paradigms may not produce the same results in terms of instrumentation overhead reduction.

An additional shortcoming is that, for smaller projects, DCC may produce worse time execution results when compared with SFL. This is due to the fact that these projects tend to produce denser program spectra matrices, because each test covers a considerable portion of the source

Dynamic Code Coverage

code. Thus, the filtering operations will be ineffective as many components will have similar fault coefficients and the overhead of re-executing the majority of a project's test suite will be greater than instrumenting the project at a high level of detail.

Dynamic Code Coverage

Chapter 4

Tooling

In this chapter, the chosen tool to host the [DCC](#) prototype – GZoltar – will be presented. Afterwards, some modifications and improvements made to the GZoltar tool will be detailed, as well as some of the most relevant implementation details of the [DCC](#) prototype.

4.1 GZoltar Toolset

GZoltar¹, also mentioned in [Section 2.2.4](#), is an Eclipse plugin that performs fault localization tasks using state-of-the-art fault localization algorithms.

The GZoltar toolset implements [SFL](#) with the Ochiai similarity coefficient (see [Equation 2.1](#)), one of the best coefficients for this purpose. GZoltar also creates powerful and navigable diagnostic report visualizations, such as Treemap and Sunburst [[Rib11](#)] (see [Figure 4.1](#)).

GZoltar is aimed at testing Java projects that use JUnit as their testing framework. Being an Eclipse (one of the most popular [IDEs](#)) plugin means that GZoltar can use many of the [IDE](#)'s functionalities, such as detection of open projects in the workspace and improved interaction between the diagnostic report visualizations and the code editor (*e.g.*, when a user clicks a certain line of code in the visualization, the corresponding file is opened in the editor, and the cursor is positioned in the desired line).

Besides fault localization, GZoltar also provides a test suit reduction and prioritization tool, coined RZoltar (see [Figure 4.2](#)). This tool minimizes the size of the original test suite using constraint-based approaches [[YH10](#)], while still guaranteeing the same code coverage. Also, RZoltar allows the user to prioritize the minimized test suites by cardinality or by execution time.

¹The GZoltar toolset can be found online at <http://gzoltar.com/>

Tooling

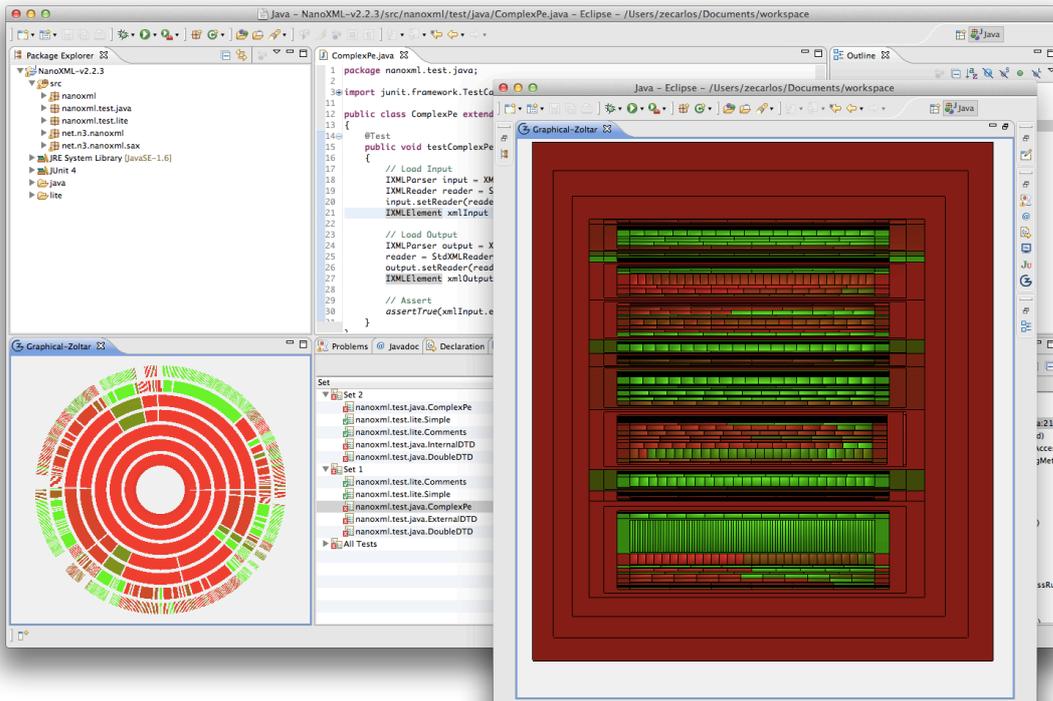


Figure 4.1: GZoltar's visualizations: Sunburst and Treemap.

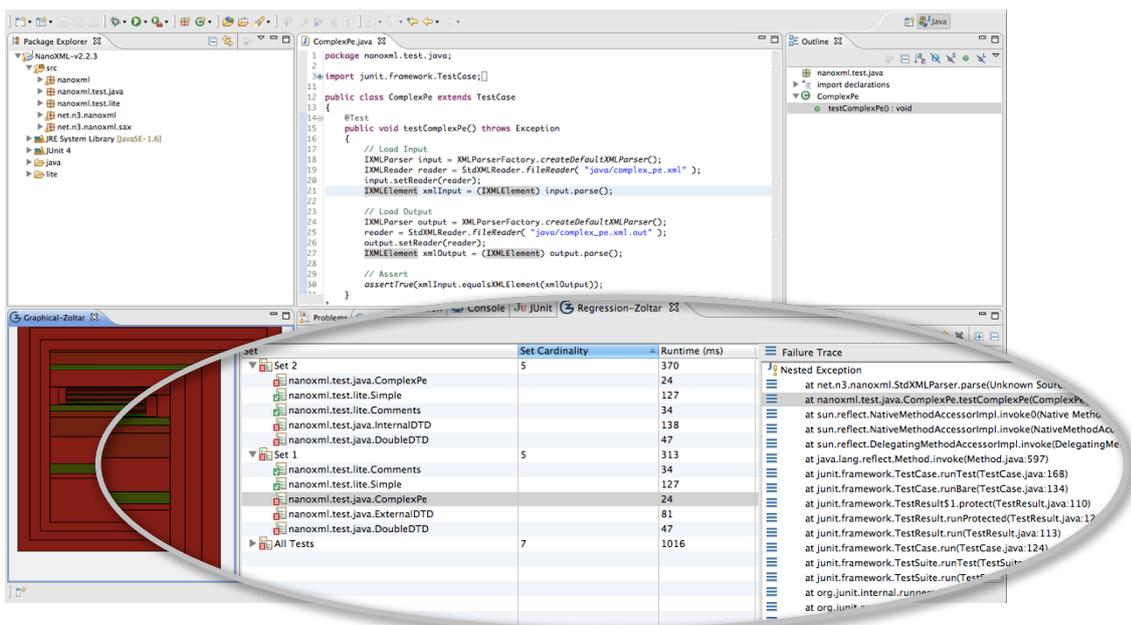


Figure 4.2: RZoltar's interface.

4.2 Modifications and Improvements

In order to implement the **DCC** prototype on top of GZoltar, some improvements and modifications had to be made to the underlying tool.

The first considerable change in GZoltar’s architecture was the test case execution. Originally, test cases were run in the same Java Virtual Machine (**JVM**) that hosted Eclipse and all its plugins. This has several disadvantages. Firstly, every class in the project had to be explicitly loaded before running any test, because the classes were not visible in the **JVM**’s *classpath*. Secondly, since the **JVM** was already running, some parameters could not be customized (*e.g.*, the maximum heap size that can be allocated). Lastly, as the **JVM**’s working directory is not the same as the system being tested, file operations with relative path names would not work correctly.

These issues were resolved by creating an external process that would spawn a new **JVM** in the **SUT**’s working directory, with its corresponding *classpath*. After running the test cases, the test results and coverage are sent to GZoltar via a socketed connection.

Another issue with the original version of GZoltar is the way how the code instrumentation is handled. GZoltar used the code coverage library JaCoCo² to obtain the coverage traces (*i.e.*, hit spectra) needed to perform the fault localization analysis. However, JaCoCo uses a **LOC** level of instrumentation, and is unable to gather coverage information without this fine-grained level of detail. To prevent this, we discarded JaCoCo and created a coverage tool able to gather traces from three different granularities: (1) class level, (2) method level and (3) **LOC** level.

This coverage tool makes use of the *java.lang.instrument* framework that was incorporated into the version 1.5 of the **JVM**. This framework allows the user to attach an agent that is able to intercept and modify a class’ *bytecode* before it is loaded by the Java *ClassLoader*. This way, the instrumentation code, written with the aid of ASM³, a Java *bytecode* manipulation framework, can be inserted at each class load time.

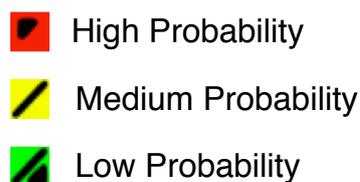


Figure 4.3: Statement failure probability markers.

Finally, as an accessibility improvement, GZoltar can now generate a list of markers that are placed on the code editor’s vertical ruler, indicating the respective line of code’s probability of being faulty when hovering the mouse. These markers, as we can see in Figure 4.3, can be of three different types: (1) red for the top third statements most likely to contain a fault, (2) yellow for the middle third statements, and (3) green for the bottom third statements. Every marker also has an

²JaCoCo – <http://www.eclemma.org/jacoco/index.html>

³ASM – <http://asm.ow2.org/>

embedded ColorADD⁴ symbol, in order to help colorblind people distinguish between markers. These annotation markers are also displayed on the Eclipse “Problems” view.

4.3 Dynamic Code Coverage Prototype

In this section, some of the most relevant implementation details of the **DCC** prototype are presented.

The instrumentation for coverage gathering for each granularity level works as follows:

- Class level – an *INVOKESTATIC* instruction is inserted at the beginning of the class initialization method (also known as *<clinit>*), every constructor method, and every public static method. The inserted instruction will call a publicly available method called *logClass* with the class details as parameters (*i.e.*, name and package). This method will register that the class was touched by the execution.
- Method level – every method of a certain class will be inserted with an *INVOKESTATIC* instruction that calls *logMethod*. This function takes as parameters the method information, namely its class, package and signature, and registers that the corresponding method was executed.
- **LOC** level – similarly to the granularities detailed previously, an invoke instruction is also inserted. In this granularity level, the instrumented instructions are placed at the beginning of each line of code, and call *logLine* (that registers a hit whenever the line is executed). One thing to note is, similarly to other code coverage tools, line coverage information can only be gathered if the classes are compiled with debug information enabled, so that source line tables that map each line to their corresponding *bytecode* instructions are available.

Some constructs, regardless of the set granularity level, are ignored while performing the instrumentation. Such is the case of synthetic constructs, which are introduced by the compiler and do not have a corresponding construct in the source code [GJSB05, p. 338]. Synthetic methods are generated for various purposes, e.g., to create bridge methods to ensure that type erasures and covariant return types are handled correctly.

Another key implementation detail worth mentioning is how the component filtering is handled. Filtering is one of the most important steps to ensure that the **DCC** algorithm performs well. Because **DCC** is a new concept, new filters should be easy to create and to replace, so that we can analyze their impact and quickly make adjustments in order to fine tune the algorithm’s performance. As such, filters use a strategy pattern [GHJV95, pp. 315-323], and can easily be interchanged.

⁴ColorADD color identification system – <http://coloradd.net/>

4.4 Discussion

Some improvements and modifications were made to the GZoltar toolset, namely to provide more control over the testing and instrumentation tasks. It is worth to note that all improvements presented in Section 4.2 have already been deployed to the development branch of GZoltar. In fact, the newest version of the tool (version 3.2.0, as of this writing) already contains all these modifications.

With these improvements, the creation of a DCC prototype was made possible. In the next chapter, the validity of the DCC approach will be evaluated using the implementation detailed in the previous sections.

Tooling

Chapter 5

Empirical Evaluation

In this chapter, we evaluate the validity and performance of the **DCC** approach for real projects. First, we introduce the programs under analysis and the evaluation metrics. Then, we discuss the empirical results and finish this section with a threats to validity discussion.

5.1 Experimental Setup

For our empirical study, four subjects written in Java were considered:

- NanoXML¹ – a small XML parser.
- `org.jacoco.report` – report generation module for the JaCoCo² code coverage library.
- XML-Security – a component library implementing XML signature and encryption standards. This library is part of the Apache Santuario³ project.
- JMeter⁴ – a desktop application designed to load test functional behavior and measure performance of web applications.

The project details of each subject are in Table 5.1. The **LOC** count information was gathered using the metrics calculation and dependency analyzer plugin for Eclipse Metrics⁵. Test count and coverage percentage were collected with the Java code coverage plugin for Eclipse EclEmma⁶.

To assess the efficiency and effectiveness of **DCC** the following experiments were performed, using fifteen faulty versions per subject program. We injected one fault in each of the 15 versions:

- **SFL** without **DCC**. This is the reference baseline.
- **DCC** with constant value coefficient filters from 0 to 0.95, with intervals of 0.05.

¹NanoXML – <http://devkix.com/nanoxml.php>

²JaCoCo – <http://www.eclEmma.org/jacoco/index.html>

³Apache Santuario – <http://santuario.apache.org/>

⁴JMeter – <http://jmeter.apache.org/>

⁵Metrics – <http://metrics.sourceforge.net/>

⁶EclEmma – <http://www.eclEmma.org/>

Empirical Evaluation

Subject	Version	LOCs (M)	Test Cases	Coverage
NanoXML	2.2.6	5393	8	53.2%
org.jacoco.report	0.5.5	5979	225	97.2%
XML-Security	1.5.0	60946	461	59.8%
JMeter	2.6	127359	593	34.2%

Table 5.1: Experimental Subjects.

- **DCC** with percentage filters from 100% to 5%, with intervals of 5%.

The metrics gathered were execution time, the size of the fault localization report, and the average LOCs needed to be inspected until the fault is located. The latter metric can be calculated by sorting the fault localization report by the value of the coefficient, and finding the injected fault’s position. In this metric, we are assuming that the developer performs the inspection in an ordered manner, starting from the highest fault coefficient LOCs.

As spectrum-based fault localization creates a ranking of components in order of likelihood to be at fault, we can retrieve how many components we still need to inspect until we hit the faulty one. Let $d \in \{1, \dots, K\}$, where K is the number of ranked components and $K \leq M$, be the index of the statement that we know to contain the fault. For all $j \in \{1, \dots, M\}$, let s_j . Then the ranking position of the faulty statement is given by

$$\tau = \frac{|\{j|s_j > s_d\}| + |\{j|s_j \geq s_d\}| - 1}{2} \quad (5.1)$$

$|\{j|s_j > s_d\}|$ counts the number of components that outrank the faulty one, and $|\{j|s_j \geq s_d\}|$ counts the number of components that rank with the same probability as the faulty one plus the ones that outrank it.

We define quality of diagnosis as the effectiveness to pinpoint the faulty component. As said before, this metric represents the percentage of components that need not be considered when searching for the fault by traversing the ranking. It is defined as

$$\left(1 - \frac{\tau}{K_{SFL}}\right) \cdot 100\% \quad (5.2)$$

where K_{SFL} is the number of ranked components of **SFL** without **DCC** – the reference baseline.

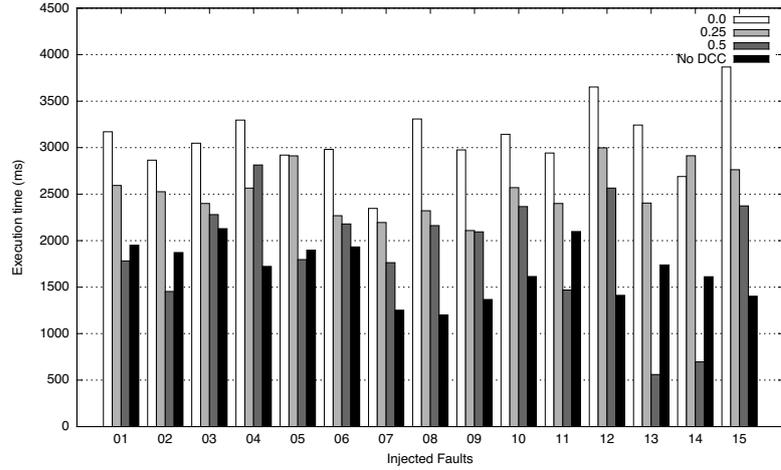
The experiments were run on a 2.7 GHz Intel Core i7 MacBook Pro with 4 GB of RAM, running OSX Lion.

5.2 Experimental Results

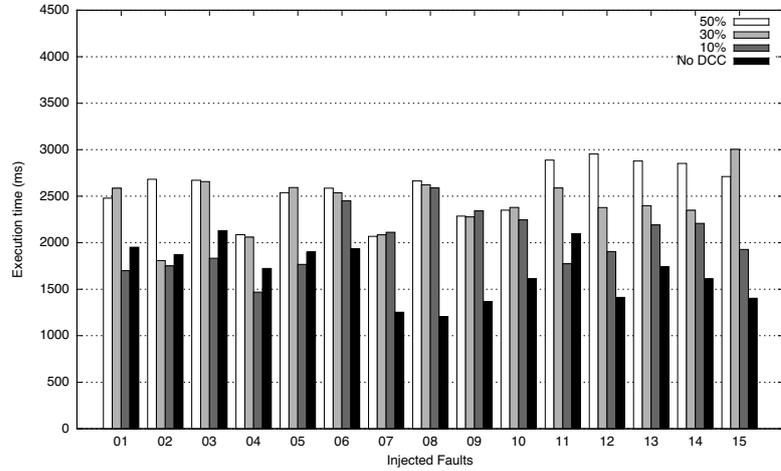
Figures 5.1, 5.2, 5.3 and 5.4 summarize the overall execution time outcomes for all the experimental subjects.

Each figure contains two plots, detailing the fault localization execution of each injected fault with **DCC** using constant coefficient value filters and with **DCC** using percentage filters, respectively. These filtering methods were previously detailed in Section 3.2.

Empirical Evaluation



(a) Coefficient filter



(b) Percentage filter

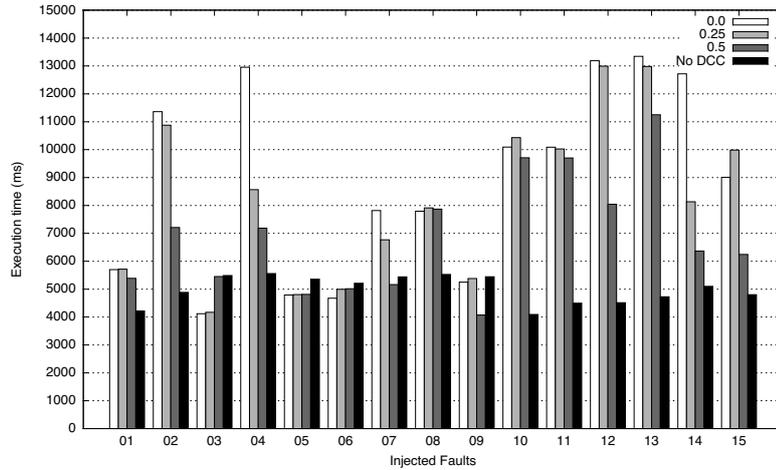
Figure 5.1: NanoXML time execution results.

Due to space constraints, only three thresholds are shown for both filters: 0.0, 0.25 and 0.5 thresholds for the constant coefficient value filters (C_f) and 50%, 30% and 10% thresholds for the percentage filters (P_f). To obtain a better understanding of the performance of each experiment, we also added, for every injected fault, the fault localization execution time of the SFL without DCC approach, labeled “No DCC” in the aforementioned figures. This way, DCC approaches can be easily compared with the SFL approach. Unless stated otherwise, every fault localization execution is able to find the injected fault (*i.e.*, the resulting report contains the injected fault).

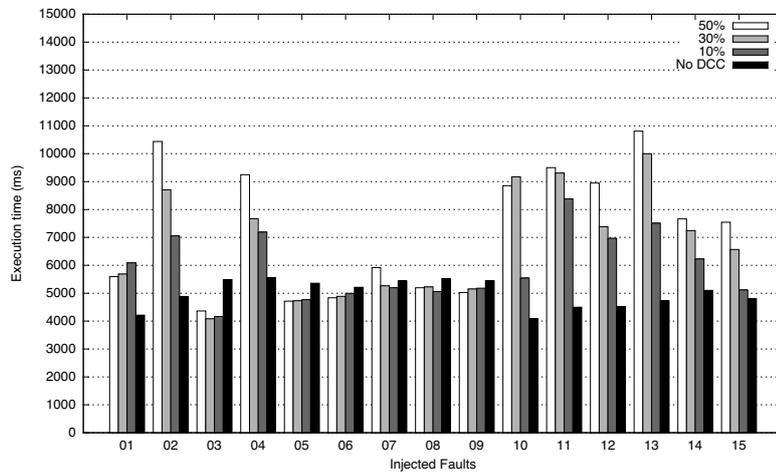
The first experimental subject to be analyzed was the NanoXML project, whose experiment results can be seen in Figure 5.1. Note that experiments 01, 11 and 13 for $C_f = 0.5$ (see Figure 5.1a) and the experiments 01, 04, 11, 12, 13, 14, 15 for $P_f = 10\%$ (see Figure 5.1b) were not able to find the injected faults.

As we can see from the experiment results, the DCC approach underperforms the current SFL method based in the execution time. Such results can be explained if we analyze the NanoXML

Empirical Evaluation



(a) Coefficient filter



(b) Percentage filter

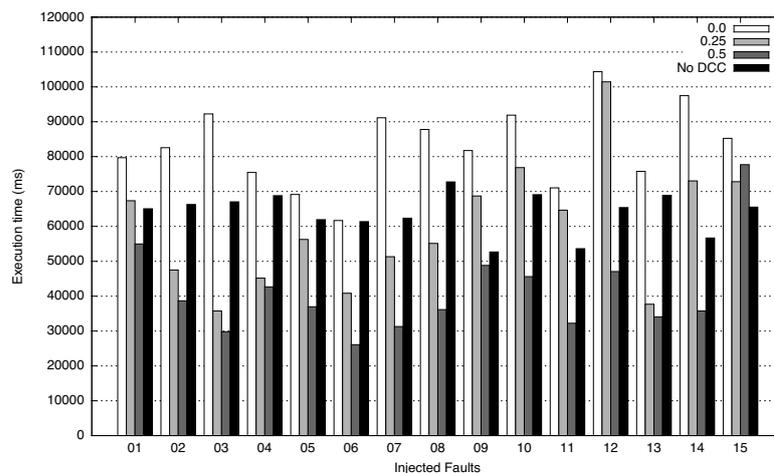
Figure 5.2: `org.jacoco.report` time execution results.

project information in Table 5.1. This project, not only is rather small in size, but also has very few test cases. At the same time, it has a coverage of over 50%. What this means is that some test cases, if not all, touch many different statements. As such, the generated program spectra matrices, detailed in Section 2.2 will be rather dense. Because of this, many components will have similar coefficients, rendering the filtering operation ineffective: either discarding many different components, or keeping a lot of components to be re-instrumented and re-tested.

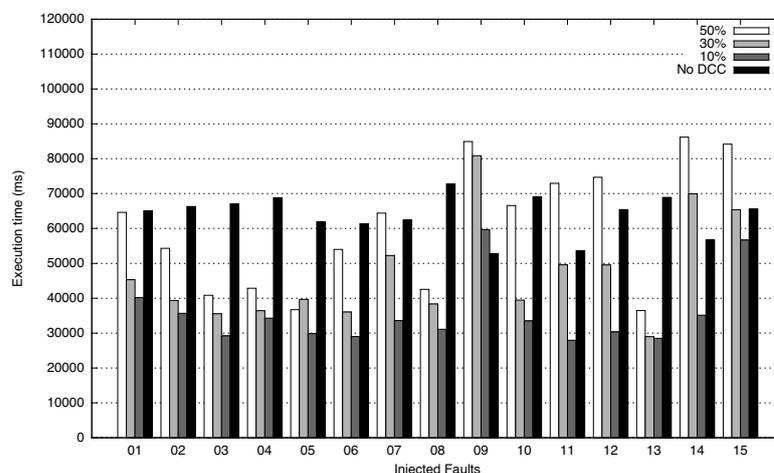
The next analyzed subject was `org.jacoco.report`, part of the JaCoCo project. The filters $C_f = 0.5$ (see Figure 5.2a) and $P_f = 10\%$ (see Figure 5.2b) were both not able to find the injected faults in experiments 09 and 15. Also, the injected fault in experiment 02 was not found in $P_f = 10\%$.

This subject, despite having many more test cases than the previous project, still has some performance drops in some of the experiments. Upon investigating the fault localization reports of the lower performance experiments, we realized that their length can be as high as 950 statements

Empirical Evaluation



(a) Coefficient filter



(b) Percentage filter

Figure 5.3: XML-Security time execution results.

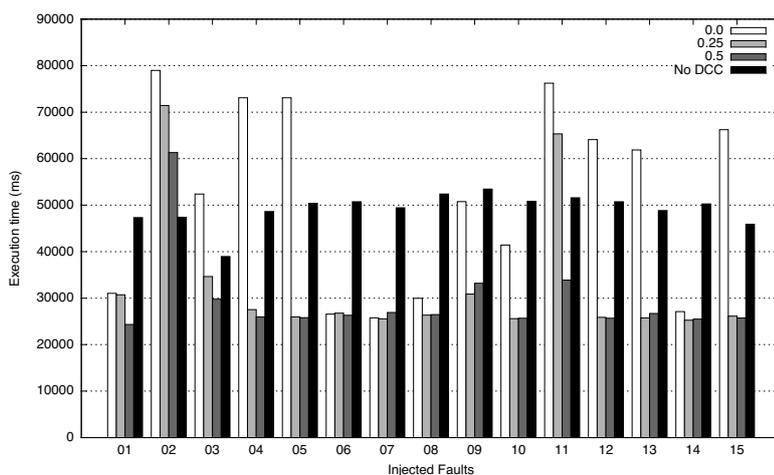
in some experiments. This means that the set of test cases that touch the injected faulty statements can cover roughly 15% of the entire project. Because of this, the same thing as the previous project happens: many components will have similar coefficients, rendering the expansions ineffective.

The following subject was the XML-Security project. Injected faults in experiments 03 and 08 were not found by $C_f = 0.5$ (see Figure 5.3a). Experiment 08 also did not have its injected fault in $P_f = 10\%$ (see Figure 5.3b).

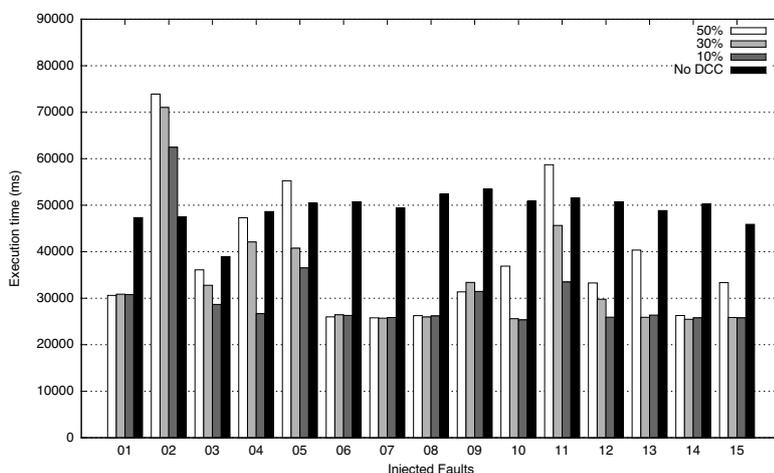
The last subject was the JMeter project. Injected faults were not found by $C_f = 0.5$ (see Figure 5.4a) in experiments 01 and 11. Every fault was found with the percentage filters.

Both XML-Security and JMeter have better results when utilizing DCC. There are mainly two reasons for these results. The first is the fact that the program spectra matrix is sparser. The other important reason is, as programs grow in size, the overhead of a fine-grained instrumentation, used in methodologies such as SFL, is much more noticeable. In this kind of sizable projects (see project informations in Table 5.1), and if the matrix is sparse enough, it is preferable to re-run

Empirical Evaluation



(a) Coefficient filter



(b) Percentage filter

Figure 5.4: JMeter time execution results.

some of the tests, than to instrument every **LOC** at the start of the fault localization process.

These time execution results confirm our assumptions that **DCC** can over-perform **SFL** for larger projects, where the instrumentation overhead is heavily noticeable. In contrast, for smaller projects, **DCC** does suffer in performance, mainly due to the fact that the overhead of re-running tests produces a bigger performance hit than the instrumentation granularity overhead. In fact, if we take into account all experiments for all four projects, there actually is an increase of execution time of 8% ($\sigma = 0.48$)⁷. However, if we only consider the larger projects where instrumentation is a more prevalent issue (*i.e.* XML-Security and JMeter), the dynamic code coverage approach can reduce execution time by 27% on average ($\sigma = 0.28$).

The other gathered metrics in this empirical evaluation, unlike execution time, show a consistent improvement over **SFL** in every project. In average, the **DCC** approach reduced 63% ($\sigma =$

⁷We have chosen to use the metrics gathered by the $P_f = 30\%$ filter since it is the best performing filter of those considered in this section that is able to find the injected faults for every experiment.

0.30) the generated fault localization ranking, providing a more concise report when compared to [SFL](#). The quality of diagnosis, described in equation 5.2, also suffered a slight improvement, from 85% ($\sigma = 0.20$) without [DCC](#) to 87% ($\sigma = 0.19$) with [DCC](#).

5.3 Threats to Validity

The main threat to external validity of these empirical results is the fact that only four test subjects were considered. Although the subjects were all real, open source software projects, it is plausible to assume that a different set of subjects, having inherently different characteristics, may yield different results. Other threat to external validity is related to the injected faults used in the experiments. These injected faults, despite being fifteen in total for each experimental subject, may not represent the entire conceivable software fault spectrum. We are also assuming that the experimental subjects do not have any faults besides those that are injected by us, and that their test cases were correctly formulated and implemented.

Threats to internal validity are related to some fault in the [DCC](#) implementation, or any underlying implementation, such as [SFL](#) or even the instrumentation for gathering program spectra. To minimize this risk, some testing and individual result checking were performed before the experimental phase.

Empirical Evaluation

Chapter 6

Conclusions and Future Work

Currently, the most effective debugging tools are Statistical Debugging Tools (**SDTs**). During the state of the art study, some flaws regarding efficiency were presented. Afterwards, a solution was proposed to minimize these flaws – the Dynamic Code Coverage (**DCC**) approach and an empirical evaluation was performed to assess the validity of the proposed solution.

6.1 State of the art of Debugging Tools

After studying the currently available debugging methodologies, **SDTs** stand out as the most effective tools to address fault localization. Other tools required more effort by the user and a more in-depth knowledge of the System Under Test (**SUT**) in order to debug it. In fact, with **SDTs**, minimal knowledge of the **SUT** is required to locate faults.

SDTs also help to improve the software development process by automating fault localization and thus enabling regression testing.

Some inefficiencies were found in current **SDT** implementations, mainly because of the instrumentation overhead required to address fault localization. This would particularly affect large scale projects, because of their high quantity of Lines Of Code (**LOCs**) that need to be instrumented.

Other debugging techniques were also studied, namely model-based reasoning techniques, such as Model-Based Software Debugging (**MBSD**). **MBSD** can produce more accurate diagnostic hypotheses when compared with statistical approaches. However, the computational effort required to create a formal model of a large, real world software application remains highly prohibitive.

6.2 Proposed Solution

The solution proposed in this thesis tries to avoid the **LOC** level of instrumentation detail in fault localization, while still using statistical debugging techniques.

Conclusions and Future Work

This approach, named **DCC**, would start by using a coarser granularity of instrumentation and progressively increasing the instrumentation detail of certain components, based on intermediate results provided by the Spectrum-based Fault Localization (**SFL**) technique.

In order to assess the validity of our approach, we have conducted an empirical evaluation on four real world open-source software projects. In each project, we have injected 15 different faults, and performed the fault localization task with **SFL** and **DCC**. With the empirical evaluation results we have demonstrated that, for large projects, our approach not only reduces the execution time by 27% on average, but also reduces the number of components reported to the user by 63% on average.

Let us revisit the research question presented in Section 1.4:

- How can a fault localization approach that instruments less software components obtain similar diagnostic results when compared with **SFL**, while reducing execution overhead?

We can conclude that our approach meets the requirements of our research question. **DCC** is a fault localization methodology that, by employing a dynamic, iterative approach, is able to instrument less software components when compared to other statistical fault localization techniques. Furthermore, the fault localization diagnostic reports generated by this methodology are often times shorter than **SFL** and the execution time is also significantly reduced when **DCC** is employed.

6.3 Main Contributions

This thesis makes the following main contributions:

1. **DCC**, a technique that automatically decides the instrumentation granularity for each module in the system, has been proposed; and
2. An empirical study to validate the proposed technique, demonstrating its efficiency using real-world, large programs. The empirical results shows that **DCC** can indeed decrease the overhead imposed in the software under test, while still maintaining the same diagnostic accuracy as current approaches to fault localization. **DCC** also decreases the diagnostic report size when compared to traditional **SFL**.

To the best of our knowledge, our dynamic code coverage approach has not been described before.

6.4 Publications

The **DCC** algorithm and the GZoltar tool have also had some exposure in academia. With the work done in this thesis, we were able to submit to the following conferences:

- Alexandre Perez, André Ribeiro and Rui Abreu. **Fault Localization using Dynamic Code Coverage** – submitted and accepted into *The 5th Meeting of Young Researchers of University of Porto (IJUP'12)*, 2012. This paper outlines the idea of using multiple levels of detail to mitigate instrumentation overhead in fault localization tools that use [SFL](#).
- José Campos, Alexandre Perez, André Ribeiro and Rui Abreu. **GZoltar: an Eclipse plugin for Testing and Debugging** – submitted and accepted into *The 27th IEEE/ACM International Conference on Automated Software Engineering (ASE'12) – Tool Demonstration*, 2012. This tool demonstration aims to present the GZoltar toolset, as well as its underlying architecture.

As of this writing, another paper is being prepared for submission to *The 6th IEEE International Conference on Software Testing, Verification and Validation (ICST'13)*. This paper is authored by Alexandre Perez, André Ribeiro and Rui Abreu. The publication will describe the [DCC](#) algorithm, and present an empirical evaluation of its performance compared to traditional [SFL](#). All of the publications mentioned above are compiled in [Appendix A](#).

6.5 Future Work

Although the initial goals of this work were achieved, there are various subjects worth researching in order to further improve the [DCC](#) fault localization methodology. Some aspects of the dynamic code coverage technique that still require further investigation are presented in the following paragraphs.

One subject worth investigating is the way of how the initial system granularity is established. Currently, this value is set manually and is the same across the entire system under test. A way to change this would be by using static analysis to assess program information and to adjust the system's initial granularity accordingly. Another approach would be to learn what were the most frequently expanded components from previous executions, and change these components' initial granularity independently.

Other issue that requires further investigation pertains to the filtering methods. It is possible that there are better filtering methods than the ones presented in this paper, namely methods that employ dynamic strategies, that change the cutting threshold based on program spectra analysis.

Our empirical evaluation results were gathered by injecting single faults. While good results have been observed, they may not represent the real world software development environment, where multiple faults may arise. Further investigation is needed, then, to evaluate [DCC](#) performance when tackling multiple simultaneous software faults.

One may also explore the ability that [DCC](#) has at stopping exploration at any desired granularity level. This methodology could be combined, then, with complementary fault localization methodologies, such as model-based reasoning approaches to debugging. As an example, [DCC](#) can be employed to obtain the top ranked methods in a program, and then the [MBSD](#) technique can be used only on these software components. [DCC](#) would be acting as a pruning mechanism,

Conclusions and Future Work

restricting the exploration performed by [MBSD](#), and significantly reducing the amount of computation required by this model-based technique.

References

- [Abr09] Rui Abreu. *Spectrum-based Fault Localization in Embedded Software*. PhD thesis, Delft University of Technology, 2009.
- [ALRL04] Algirdas Avižienis, Jean-Claude Laprie, Brian Randell, and Carl E. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable Secure Computing*, 1(1):11–33, 2004.
- [AZG06] Rui Abreu, Peter Zoetewij, and Arjan J. C. van Gemund. An evaluation of similarity coefficients for software fault localization. In *Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing*, PRDC '06, pages 39–46, Washington, DC, USA, 2006. IEEE Computer Society.
- [AZGV09] Rui Abreu, Peter Zoetewij, Rob Golsteijn, and Arjan J C Van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, 82(11):1780–1792, 2009.
- [AZvG07] Rui Abreu, Peter Zoetewij, and Arjan J. C. van Gemund. On the accuracy of spectrum-based fault localization. In *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, pages 89–98, Washington, DC, USA, 2007. IEEE Computer Society.
- [AZvG09] Rui Abreu, Peter Zoetewij, and Arjan J. C. van Gemund. Spectrum-based multiple fault localization. In Gabriele Taentzer and Mats Heimdahl, editors, *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE'09)*, Auckland, New Zealand, 16 – 20 November 2009. IEEE Computer Society, to appear.
- [CLR05] M L Corliss, E C Lewis, and A Roth. Low-Overhead Interactive Debugging via Dynamic Instrumentation with DISE. *11th International Symposium on High-Performance Computer Architecture*, pages 303–314, 2005.
- [dKW87] Johan de Kleer and Brian C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130, 1987.
- [Dow97] Mark Dowson. The Ariane 5 software failure. *SIGSOFT Software Engineering Notes*, 22(2):84–, March 1997.
- [FSW96] Gerhard Friedrich, Markus Stumptner, and Franz Wotawa. Model-based diagnosis of hardware designs. In Wolfgang Wahlster, editor, *Proceedings of the 12th European Conference on Artificial Intelligence (ECAI'96)*, pages 491–495, Budapest, Hungary, 11–16 August 1996. John Wiley and Sons, Chichester.

REFERENCES

- [FSW99] Gerhard Friedrich, Markus Stumptner, and Franz Wotawa. Model-based diagnosis of hardware designs. *Artificial Intelligence*, 111(1-2):3–39, 1999.
- [GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995.
- [GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley, 2005.
- [GKM82] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. Gprof: A call graph execution profiler. *SIGPLAN Not.*, 17:120–126, June 1982.
- [GvVEB06] D. Graham, E. van Veenendaal, I. Evans, and R. Black. *Foundations of Software Testing: ISTQB Certification*. Cengage Learning Business Press, 1st edition, 2006.
- [HRS⁺00] Mary Jean Harrold, Gregg Rothermel, Kent Sayre, Rui Wu, and Lui Yi. An empirical investigation of the relationship between fault-revealing test behavior and differences in program spectra. *STVR Journal of Software Testing, Verification, and Reliability*, (3):171–194, September 2000.
- [HS02] B Hailpern and P Santhanam. Software debugging, testing, and verification. *IBM Systems Journal*, 41(1):4–12, 2002.
- [JAvG09a] Tom Janssen, Rui Abreu, and Arjan J.C. van Gemund. Zoltar: A spectrum-based fault localization tool. In *Proceedings of the 2009 ESEC/FSE workshop on Software integration and evolution @ runtime*, SINTER '09, pages 23–30, New York, NY, USA, 2009. ACM.
- [JAvG09b] Tom Janssen, Rui Abreu, and Arjan J.C. van Gemund. Zoltar: A Toolset for Automatic Fault Localization. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 662–664. IEEE, November 2009.
- [JD88] Anil K. Jain and Richard C. Dubes. *Algorithms for clustering data*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.
- [JH05] James A. Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ASE '05, pages 273–282, New York, NY, USA, 2005. ACM.
- [JHS02] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 467–477, New York, NY, USA, 2002. ACM.
- [Kid98] P.A. Kidwell. Stalking the elusive computer bug. *Annals of the History of Computing*, *IEEE*, 20(4):5–9, oct-dec 1998.
- [KM08] Andrew J. Ko and Brad A. Myers. Debugging reinvented: asking and answering why and why not questions about program behavior. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 301–310, New York, NY, USA, 2008. ACM.

REFERENCES

- [LFY⁺06] C. Liu, L. Fei, X. Yan, J. Han, and S.P. Midkiff. Statistical debugging: A hypothesis testing-based approach. *IEEE Transactions on Software Engineering (TSE)*, 32(10):831–848, 2006.
- [Lio96] J. L. Lions. Ariane 5: Flight 501 failure. Technical report, ESA: Ariane 501 Inquiry Board, 1996.
- [MS07] W Mayer and M Stumptner. Model-Based Debugging – State of the Art And Future Challenges. *Electronic Notes in Theoretical Computer Science*, 174(4):61–82, 2007.
- [MS08] W. Mayer and M. Stumptner. Evaluating Models for Model-Based Debugging. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, ASE '08*, pages 128–137, Washington, DC, USA, 2008. IEEE Computer Society.
- [NW08] Mihai Nica and Franz Wotawa. From constraint representations of sequential code and program annotations to their use in debugging. In Malik Ghallab, Constantine D. Spyropoulos, Nikos Fakotakis, and Nikolaos M. Avouris, editors, *Proceedings of the 18th European Conference on Artificial Intelligence (ECAI'08)*, volume 178 of *Frontiers in Artificial Intelligence and Applications*, pages 797–798, Patras, Greece, 21–26 July 2008. IOS Press.
- [RA10] André Ribeiro and Rui Abreu. The gzoltar project: a graphical debugger interface. In *Proceedings of the 5th international academic and industrial conference on Testing - practice and research techniques, TAIC PART'10*, pages 215–218, Berlin, Heidelberg, 2010. Springer-Verlag.
- [RAR11] André Ribeiro, Rui Abreu, and Rui Rodrigues. An OpenGL-based eclipse plug-in for visual debugging. In *Proceedings of the 1st Workshop on Developing Tools as Plug-ins, TOPI '11*, pages 60–60, New York, NY, USA, 2011. ACM.
- [RBDL97] Thomas Reps, Thomas Ball, Manuvir Das, and James Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In *Proceedings of the 6th European Software Engineering conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of software engineering, ESEC '97/FSE-5*, pages 432–449, New York, NY, USA, 1997. Springer-Verlag New York, Inc.
- [Rib11] André Ribeiro. *GZoltar: A Graphical Debugger Interface*. MSc Thesis, Faculdade de Engenharia da Universidade do Porto, 2011.
- [Rog02] S Rogerson. The chinook helicopter disaster. *The Institute for the Management of Information Systems (IMIS)*, 12(2), 2002.
- [Ros95] D.S. Rosenblum. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, 21(1):19–31, Jan 1995.
- [SPS06] Richard Stallman, Roland Pesch, and Stan Shebs. *Debugging with gdb*. Free Software Foundation, 2006.
- [Tas02] G Tassey. The Economic Impacts of Inadequate Infrastructure for Software Testing, 2002.

REFERENCES

- [TH02] Mustafa M. Tikir and Jeffrey K. Hollingsworth. Efficient instrumentation for code coverage testing. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '02, pages 86–96, New York, NY, USA, 2002. ACM.
- [WSM02] Franz Wotawa, Markus Stumptner, and Wolfgang Mayer. Model-based debugging or how to diagnose programs automatically. In T. Hendtlass and M. Ali, editors, *Proceedings of IAE/AIE 2002*, volume 2358 of *LNCS*, pages 746–757, Cairns, Australia, 17 – 20 June 2002. Springer-Verlag.
- [WWQZ08] Eric Wong, Tingting Wei, Yu Qi, and Lei Zhao. A crosstab-based statistical method for effective fault localization. In Rob Hierons and Aditya Mathur, editors, *Proceedings of the 1st International Conference on Software Testing, Verification, and Validation (ICST'08)*, pages 42–51, Lillehammer, Norway, 9 – 11 April 2008. IEEE Computer Society.
- [YH10] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: A survey. *Software Testing, Verification and Reliability*, 2010.
- [YLW06] Qian Yang, J. Jenny Li, and David Weiss. A survey of coverage based testing tools. In *Proceedings of the 2006 international workshop on Automation of software test*, AST '06, pages 99–103, New York, NY, USA, 2006. ACM.

Appendix A

Publications

Fault Localization using Dynamic Code Coverage

A. Perez¹, A. Riboira¹, R. Abreu¹

¹Department of Informatics Engineering, Faculty of Engineering, University of Porto, Portugal.

In software development, a large amount of resources is spent in the debugging phase [1]. This process of detecting, locating and fixing faults in the source code is not trivial and rather error-prone. In fact, 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 [2].

In order to improve the debugging efficiency, this process needs to be automated. Some effort was already made to automatically assist the detecting and locating phases. This led to the creation of automatic fault localization tools, namely Zoltar [3] and Tarantula [4]. The tools instrument the source code to obtain code coverage traces for each test, which are then analyzed to return a list of potential faulty locations. To improve the exploration and intuitiveness of that list, an Eclipse [5] plugin was also developed – GZoltar [6] – that adds fault localization functionality to an integrated development environment, with several visualization options.

Although these tools can be helpful, they do not scale. These tools need to instrument a large portion of the project at the line of code level so that an analysis can be performed. This is acceptable for small software applications, but impractical for large, real-world projects that contain hundreds of thousands of lines of code.

A new approach to this problem would be to avoid as much as possible the line of code level of instrumentation detail, while still using the proven techniques [3] that these fault localization tools implement, using dynamic code coverage.

The dynamic code coverage method consists of coarsening the granularity of the instrumentation, obtaining only coverage traces for large components, and running the same fault localization analysis detailed previously. This would return a list of potential faulty components. After that, the components most likely to contain a fault would be re-instrumented, with a finer-grained detail. The process will then loop until a list of the lines of code most likely to contain a fault was reached.

The objective of this work is to implement a working prototype of this dynamic code coverage approach in GZoltar, and evaluate its performance improvement. Another important goal is to minimize the impact of the debugging phase in the software project's resources.

References:

- [1] G. Tassely. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology RTI Project*, 2002.
- [2] Andrew J. Ko and Brad A. Myers. Debugging Reinvented : Asking and Answering Why and Why Not Questions about Program Behavior. ICSE '08.
- [3] Tom Janssen, Rui Abreu, and Arjan J. C. van Gemund. Zoltar: A toolset for automatic fault localization. ASE '09.
- [4] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. ICSE '02.
- [5] Eclipse Integrated Development Environment. <http://www.eclipse.org/>.
- [6] André Riboira and Rui Abreu. The gzoltar project: a graphical debugger interface. TAIC PART'10.

GZoltar: an Eclipse plug-in for Testing and Debugging

José Campos

Department of Informatics Engineering
Faculty of Engineering of University of Porto
Portugal

jose.carlos.campos@fe.up.pt

Alexandre Perez

Department of Informatics Engineering
Faculty of Engineering of University of Porto
Portugal

alexandre.perez@fe.up.pt

André Ribeiro

Department of Informatics Engineering
Faculty of Engineering of University of Porto
Portugal

andre.riboira@fe.up.pt

Rui Abreu

Department of Informatics Engineering
Faculty of Engineering of University of Porto
Portugal

rui@computer.org

ABSTRACT

Testing and debugging is the most expensive, error-prone phase in the software development life cycle. Automated testing and diagnosis of software faults can drastically improve the efficiency of this phase, this way improving the overall quality of the software. In this paper we present a toolset for automatic testing and fault localization, dubbed GZOLTAR, which hosts techniques for (regression) test suite minimization and automatic fault diagnosis (namely, spectrum-based fault localization). The toolset provides the infrastructure to automatically instrument the source code of software programs to produce runtime data, which is subsequently analyzed to both minimize the test suite and return a ranked list of diagnosis candidates. The toolset is a plug-and-play plug-in for the Eclipse IDE to ease world-wide adoption.

Categories and Subject Descriptors

D.2.5 [Software engineering]: Testing and Debugging

General Terms

Reliability, Experimentation

Keywords

Eclipse plug-in, Automatic Testing, Automatic Debugging, GZOLTAR, RZOLTAR

1. TESTING & DEBUGGING

Testing and Debugging is an important, yet the most expensive and tedious phase of the software development life-cycle. Although there are already off-the-shelf frameworks to ease this tasks, they still do not offer enough capabilities to fully automate this phase. Well known (unit) testing

frameworks include JUNIT, TESTNG, and JTest, which automated the test execution but do not offer capabilities for, e.g., test suite minimization based on some criteria (such as coverage). Several debugging tools exist which are based on stepping through the execution of the program (e.g., GDB and DDD). These traditional, manual fault localization approaches have a number of important limitations. The placement of print statements as well as the inspection of their output are unstructured and ad-hoc, and are typically based on the developer's intuition. In addition, developers tend to use only test cases that reveal the failure, and therefore do not use valuable information from (the typically available) passing test cases.

Aimed at drastic cost reduction, much research has been performed in developing automatic testing and fault localization techniques and tools. As far as testing is concerned, several techniques have been proposed minimize and prioritize test cases in order to reduce execution time and failure detection, while maintaining similar code coverage [14]. This paper presents a toolset, coined GZOLTAR, that provides a technique for test suite reduction and prioritization. The technique minimizes the original test suite using a novel constraint-based approach [4], while still guaranteeing the same code coverage. Furthermore, the technique allows the user to prioritize the minimized test suites by cardinality and execution time of the computed test suites.

As for debugging, one of the predominant techniques are those based on a black box statistics-based method which takes a program and available test cases and returns the most probable location (component) that explains the observed failed test cases. The GZOLTAR toolset implements a technique called spectrum-based fault localization (SFL [1]; in particular, the tool provides the Ochiai [1] incarnation of SFL, which is amongst the best for fault localization). SFL is based on instrumenting a program to keep track of executed parts. This log, run-time data is then analyzed to yield a list of source code locations ordered by the likelihood of it containing the fault. Furthermore, the toolset enables a program to be trained with expected behavior and to automatically detect an error if unexpected behavior is observed. The fact that no knowledge is needed of the program to acquire possible fault locations makes this set of tools a useful extension to currently applied methods of testing and debugging.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ASE 2012 Essen, Germany

Copyright 2008 ACM XXX-X-XXXXX-XXX-X/XX/XX ...\$10.00.

The GZOLTAR toolset, being developed at the University of Porto, aims at providing state-of-the-art techniques for (regression) test suite minimization and fault localization. The tool is available for download at <http://www.gzoltar.com>.

2. GZOLTAR TOOLSET

GZOLTAR is an Eclipse plug-in which produces accurate fault localization data using state-of-the-art spectrum-based fault localization algorithms, and provide the last studies in the field of regression testing. It also creates powerful and navigable diagnostic reports' visualizations, such as Treemap and Sunburst (see Fig. 3).

The integration on Eclipse (one of the most popular IDEs) is extremely useful as the Eclipse functionalities, such as detection of open projects in the workspace and their classes, interact with the (visual) diagnostic reports to dynamically open the code editor in a potential faulty statement, can be used by the toolset.

GZOLTAR aids developers find faults faster, thus spending less time and resources in testing and debugging. This in turn leads to a higher software reliability level and/or to a decrease of its test period, thus reducing costs significantly.

2.1 GZoltar Architecture

The GZOLTAR is mainly written in Java and also uses third-party open source programs. The Eclipse's Workspace component is used to gather information it needs, such as open projects, their classes, and JUnit tests. ASM [3], a Java bytecode engineering library, is used to instrument the System Under Test (SUT) in order to obtain coverage traces when executing the unit tests with JUnit. The Eclipse's Workbench component is used for generating the Eclipse User Interface (UI) tasks. This component has Standard Widget Toolkit (SWT) to create the GZOLTAR view, and provides a bridge to Abstract Windows Toolkit (required by the Java OpenGL (JOGL), the component that provides OpenGL bindings to Java). JOGL generates the OpenGL-based visualizations displayed on the GZoltar view. Integrated in the GZOLTAR plug-in, but written in C is the MINION constraint solver¹. Finally, the TRIE [7] structure has a interface written in Java and implemented in C for efficiency. For a schematic view of these technological layers and their interactions, see Fig. 1.



Figure 1: GZoltar Layers. Integration between GZoltar and other technologies.

¹MINION Homepage, <http://minion.sourceforge.net/>, 2012.

2.2 GZoltar Flow

The GZOLTAR processing flow can be divided into eight main stages such Fig. 2 show.

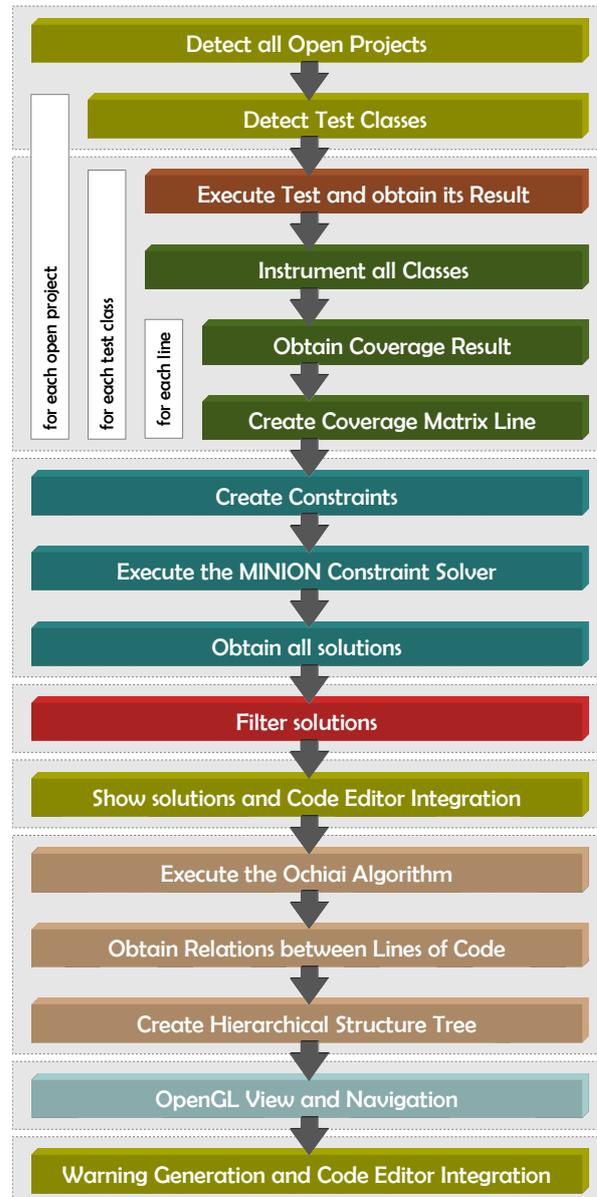


Figure 2: Information flow.

Initial Eclipse Integration: Eclipse makes it possible to automatically detect all open projects in the IDE. Once the set of the open projects is known, GZOLTAR search indexes all their classes and JUnit test classes (those who have test methods written in JUnit syntax, to be executed later). To avoid differences between source files and compiled classes, at this stage, GZOLTAR forces Eclipse to build the open projects, to guarantee that it is working with the latest version of the code.

JUnit and ASM: For each project there is a list of all test classes. For each class in the project all code is instru-

mented to allow the code coverage process. That process aims at detecting if a given line of code was executed or not. GZOLTAR uses ASM to instrument all open projects, thus being capable of debugging projects that call methods from other projects. Subsequently, test classes, implemented in the JUnit syntax, are executed automatically.

The information whether a test case has passed or failed is also stored to be used later by both the RZOLTAR (namely, to display if test fails) and GZOLTAR (namely, to compute the diagnostic report) views. The results are saved into a coverage matrix [4], a $N \times M$ binary matrix A , where N is the execution of a test case, M correspond to different components of a software program, and a_{ij} is the coverage for component j when test i is executed. Once the code coverage matrix is gathered, RZOLTAR analyzes them to minimize the suite.

Gathering code coverage information of the SUT for each test case consists of three steps: (1) the code is instrumented to register what statements where touched by an execution, (2) the test case is executed, and (3) a coverage trace of what statements where executed is computed, and appended to the coverage matrix.

In summary, at this stage, all code from open projects are instrumented and built, test classes are executed, and code coverage matrix (plus any other relevant information) is stored.

MINION: After collecting the coverage of the SUT, into the coverage matrix explained before, the coverage information is passed to the constraint solver and it returns at least one minimum set that cover the entire software program such as original set.

Filtering out Solutions: The results provided by constraint solver are then filtered out using a TRIE data structure. Essentially, this stage is to discard non-minimal test suites from the collection presented to the user.

Show Solutions: At this stage the user can select on minimum subset to re-execute, or sorted all subsets by their cardinality or execution time of each subset.

Run Ochiai: At this stage, GZOLTAR executes the SFL algorithm Ochiai, known to be amongst the best performing techniques for fault localization [1]. Based on every JUnit test result, the Ochiai similarity coefficient is calculated for every element of the system.

Graphical Visualization: To perform powerful and efficient visualizations of SUT, GZOLTAR use OpenGL technology in order to take advantage of the Graphics Processing Unit (GPU).

Warning Generation and Final Eclipse Integration: The GZOLTAR plug-in also generates warnings that integrates with the code editor, marking the lines that have a high probability of being faulty.

After all this stages, the user can inspect the fail unit tests and faulty lines (if they exist). This is a recursive process, so until all faults are not fixed, user can select a minimum set to re-execute (testing and saving time at the same time), fix and re-execute again.

2.3 Eclipse Views

By default, GZOLTAR offers two Eclipse views integrated into the IDE: GZOLTAR (Fig. 3) and RZOLTAR (Fig. 5).

While analyzing the SUT, the user can click on the visual representation of a line of code on the GZOLTAR view, and jump directly to that line in the Eclipse’s code editor. An Eclipse code editor is opened with the text cursor placed on the line selected in the GZOLTAR view. Furthermore, GZOLTAR also generates a list of markers placed on the code editor’s vertical ruler, indicating the respective line of code’s probability of being faulty when hovering the mouse. These markers, as we can see in Fig. 4, can be of three different types: (1) red for the top third statements most likely to contain a fault, (2) yellow for the middle third statements, and (3) green for the bottom third statements. Every marker also has an embedded ColorADD² symbol, in order to help colorblind people distinguish between markers. These annotation markers are also displayed on the Eclipse “Problems” view.

GZOLTAR view also provides two visualizations [13] Treemap and Sunburst, such as illustrated at Fig. 3. With this seamless integration, the user can easily analyze the SUT structure and localize the root cause of observed failures. Thus, GZOLTAR provides a easy way to access directly to the source code in order to fix faults.

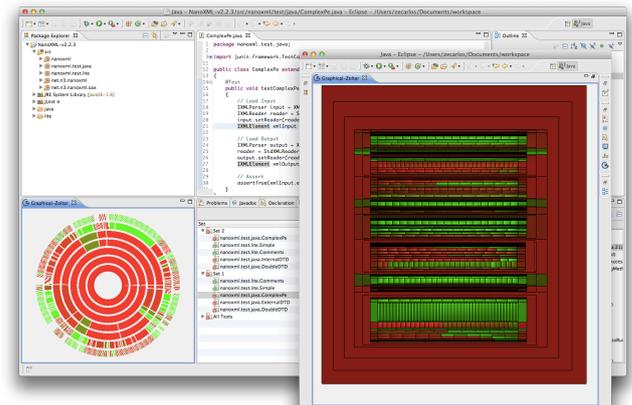


Figure 3: GZoltar’s visualizations: Treemap and Sunburst [13].

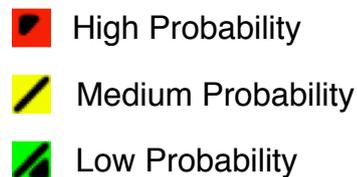


Figure 4: GZoltar’s statement failure probability markers.

The RZOLTAR view (Fig. 5) is divided in two layers. On the left layer, the user can access the list of minimum set cov-

²ColorADD color identification system, <http://coloradd.net/>, 2012.

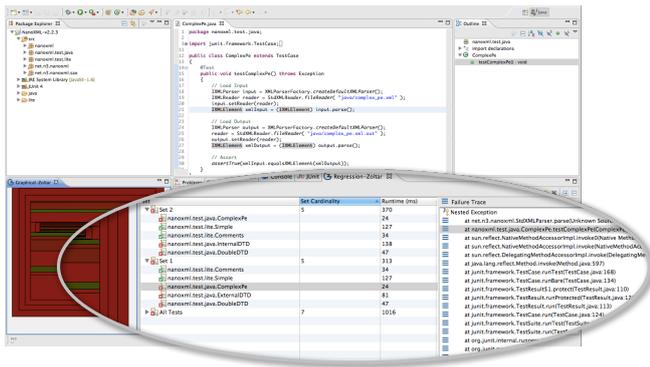


Figure 5: RZoltar interface.

erage (including the set with all test cases, just in case the user decides to re-execute the original test suite) and check the result of test cases (pass, fail, or error) by the color of icon. If test fail or return an error, the error message is displayed in “Trace” window. In all layers, the user can always double-click on a test case and jump to the test case file, or at failure trace, double-click goes to line (presents in that layer). This is similar to the functionalities offered by JUnit. The RZOLTAR view offers to criteria to prioritize test suites: *Cardinality of Set* and *Runtime*. The latter prioritization orders the minimum sets found (test suites of reduced size) using the take it takes to re-execute the suite, whereas the former orders the sets by the number of tests cases (set cardinality).

All in all, GZOLTAR provides an excellent ecosystem for regression testing and automatic debugging. This toolset is also straightforward to understand because we use familiar interface features (e.g., icons similar to the ones used in JUnit).

3. RELATED WORK

Nowadays most of the IDEs in market only offer a limited and manual debugging tool, such as breakpoints, conditional breakpoints, or the possibility to execute the software in a step-by-step. To the best of our knowledge, the most well known automatic debugging tools is Tarantula [12]. This independent tool is based on the the code coverage of multiple test executions of a given system. Although, Tarantula has not integration with any IDE, and do not support unit tests. Zoltar [11] is another available automatic debugging tool. It uses similarity coefficients to predict the failure probability of each line of code. Currently Zoltar runs only on Linux systems and is only works only with projects written in C language. Other relevant tools for automatic debugging are: Vida [8] an Eclipse plug-in based in Tarantula approach; EzUnit4 [2] is also an Eclipse plug-in that uses statistical analysis to determine the failure probability of every tested method.

Regression testing has been the field of several research studies in last years. Similar to RZOLTAR, the following tools are the best known tools for regression testing: MINTS [10] which uses an Integer Linear Programming (ILP) solver with multi-criteria test minimization; TestTube [5] partitions the SUT in several program entities, and follow the execution of test cases to analyze the relation between tests and pro-

gram entities. Other techniques for regression testing include Greedy heuristic [6] and Program Slicing [9].

4. CONCLUSIONS

Testing and debugging is tedious and cumbersome phase in the software development life-cycle. Aimed at aiding the developer to test the software application and, if needed, pinpoint the source of observed failures, this paper describes GZOLTAR, an Eclipse plug-in which has a GZOLTAR view to deal with tasks about debugging, and RZOLTAR view for regression testing purpose. The toolset as well as a tutorial can be obtained from <http://www.gzoltar.com>.

Future work include the following. We plan to provide more techniques for minimizing test suites (e.g., greedy, MINTS), add more visualizations of the diagnostic reports. Furthermore, there are plans to add the capability of dynamically instrumenting the source code, this way reducing the overhead imposed to collect information.

5. REFERENCES

- [1] R. Abreu, P. Zoetewij, R. Golsteijn, and A. Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, 82(11):1780–1792, Nov. 2009.
- [2] P. Bouillon, J. Krinke, N. Meyer, and F. Steimann. EzUnit: A Framework for Associating Failed Unit Tests with Potential Programming Errors. In G. Concas, E. Damiani, M. Scotto, and G. Succi, editors, *Agile Processes in Software Engineering and Extreme Programming*, volume 4536 of *Lecture Notes in Computer Science*, pages 101–104. Springer Berlin / Heidelberg, 2007.
- [3] E. Bruneton, R. Lenglet, and T. Coupaye. ASM: A code manipulation tool to implement adaptable systems. In *Adaptable and extensible component systems*, 2002.
- [4] J. Campos. Regression testing with GZoltar: Techniques for test suite minimization, selection, and prioritization. MSc Thesis, University of Porto, 2012.
- [5] Y.-F. Chen, D. S. Rosenblum, and K.-P. Vo. TestTube: a system for selective regression testing. In *Proc. ICSE’94, ICSE ’94*, pages 211–220, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.
- [6] V. Chvatal. A Greedy Heuristic for the Set-Covering Problem. *Mathematics of Operations Research*, 4(3):233–235, 1979.
- [7] E. Fredkin. Trie memory. *Commun. ACM*, 3:490–499, September 1960.
- [8] D. Hao, L. Zhang, L. Zhang, J. Sun, and H. Mei. VIDA: Visual interactive debugging. In *Proc. of ICSE’09, ICSE ’09*, pages 583–586, Washington, DC, USA, 2009. IEEE Computer Society.
- [9] M. J. Harrold, R. Gupta, and M. L. Soffa. A methodology for controlling the size of a test suite. *ACM Trans. Softw. Eng. Methodol.*, 2:270–285, July 1993.
- [10] H.-Y. Hsu and A. Orso. *MINTS: A general framework and tool for supporting test-suite minimization*, volume 0, pages 419–429. IEEE Computer Society, 2009.
- [11] T. Janssen, R. Abreu, and A. J. C. v. Gemund. Zoltar: A toolset for automatic fault localization. In *Proc. of ASE ’09*, pages 662–664, Washington, DC, USA, 2009. IEEE CS.
- [12] J. A. Jones, M. J. Harrold, and J. T. Stasko. Visualization for Fault Localization. In *Proc. of ICSE’01*, pages 71–75. IEEE Computer Society Press, 2001.
- [13] A. Ribeiro. GZoltar: A graphical debugger interface. MSc Thesis, University of Porto, 2011.
- [14] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: A survey. *Software Testing, Verification and Reliability*, 2010.

APPENDIX

A. TOOL PRESENTATION

To demonstrate how can we quickly test and debug a program using GZOLTAR toolset, we consider a Java program, coined NanoXML³. First we introduced a fault on NanoXML program. After that we use RZOLTAR view to determine a minimum set which we can test the program, and in an attempt to try to quickly find the failure previously introduced, we use GZOLTAR view to debug the program. The following demonstration can also be seen at <http://www.gzoltar.com/>.

A.1 NanoXML

NanoXML is a small XML parser developed in Java. This is an open source program of medium size, with 4660 lines of code and a suite of JUnit tests.

Originally all the unit tests present in the program return success. To validate if the ecosystem for testing e debugging GZOLTAR is in fact practical and helpful for developers, a fault was injected in the program with the objective of some tests now returns error. The failure results of a change in Line number 109 of the method `skipTag` at `XMLUtil` class, at package `net.n3.nanoxml`, from “`case '>';`” to “`case ']'`”.

To start, first we open the two views: GZOLTAR view and RZOLTAR view on Eclipse IDE, such as Fig. 6 show.

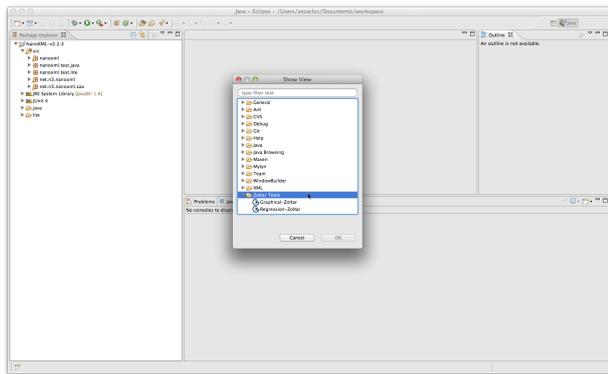


Figure 6: GZoltar and RZoltar in Eclipse’s View selection.

After this step, the GZOLTAR and RZOLTAR view are located at the bottom of Eclipse (see Fig. 7).

Now, to test all NanoXML program we need to deal with RZOLTAR view. This view is responsible for all actions about regression testing. It is divided in two layers, on the left user can access to the list of minimum set coverage (including the set with all test cases); on the right user can see “Failure Trace” of every faulty test, such as Fig. 8 show.

At RZOLTAR view there are also two implicit prioritization: *Cardinality of Set* and *Runtime*. First prioritization can order the minimum sets through the number of tests in every sets. Second prioritization can sort the minimum sets by the total of time which set needed to re-run.

Every time you change your project code, RZOLTAR have to update its information. This is not made automatically because of performance reasons. To do so, click on “Refresh”

³NanoXML Homepage, <http://devkix.com/nanoxml.php>, 2012.

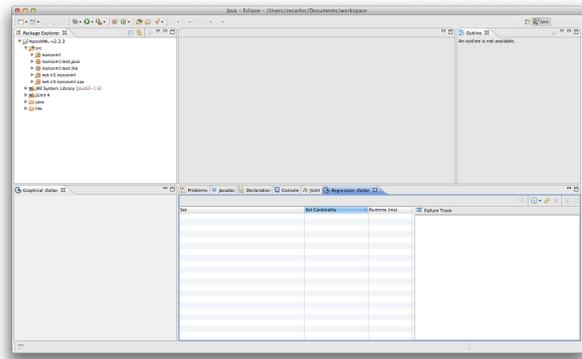


Figure 7: Default visualization of Eclipse with GZoltar plug-in installed.

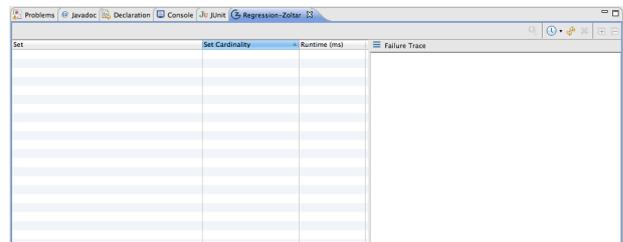


Figure 8: RZoltar view.

button (CTRL + F5) or if you already do this, you can only select a set of tests and click on Re-Run button, to run again the selected set. You will then see a view with the updated data. If you are working with big projects, it is normal if you have a delay between the time you press “Refresh” button and have the view ready to navigate. User can always clear, expand and collapse results, with respectively buttons (see Fig. 9).



Figure 9: RZoltar toolbar.

From the left to the right, in Figure 9 appears: “Re-run the selected set”, “How long are you willing to wait for the results?”, “Refresh”, “Clear Results”, “Expand” and “Collapse”. “Re-run the selected set” allows the re-execution of a selected set. “How long are you willing to wait for the results?” provides some options to user, related with time to calculate the minimum set coverage, for example, “I want to wait 30sec (max limit) for a minimum solution”, or “I want the first 100 minimum solutions”. “Refresh”, runs all unit tests presented in Java project, and calculate the minimum set coverage. “Clear Results”, erase the results showed by GZOLTAR and RZOLTAR view. “Expand” and “Collapse”, show/hide the sets in RZOLTAR view.

Wherefore, to start testing NanoXML program we need to click on “Refresh” button and after that the result of every unit test is show at RZOLTAR view (on right at Fig. 10).

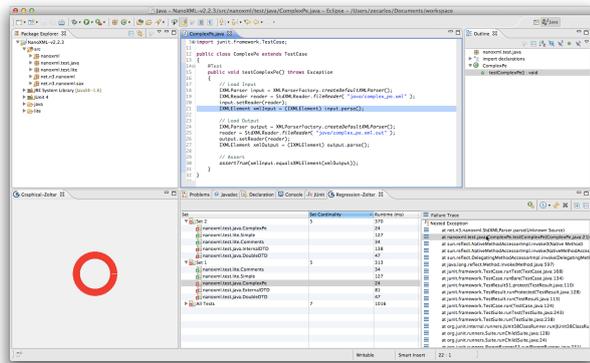


Figure 10: Result for the first execution of all unit tests.

As we can see in Fig. 10, there are some tests that fail, and on the right at “Failure Trace”, we can also investigate the returned failure of every test. On the two layers of RZOLTAR view, user can always double-click on a test case and jump to the file with test case, or at failure trace, double-click goes to line (presents in that layer).

Now, we have some test which can not return success, so we need to inspect the code or use interactive GZOLTAR view to do that.

GZOLTAR view begin with the default visualization, Sunburst (on left at Fig. 12) but pressing the key 2 we can change to TreeMap visualization (on right at Fig. 12). Sunburst gives a view that privileges more the hierarchical location, and Treemap, on its turn, presents a view that privileges more the tree leafs.

It is possible to see if a project has faults by analyzing its components colors. If a project does not fail in any test, all components will be rendered in green, otherwise, a color that varies from green to red will represent the component failure probability. It is possible to place the mouse over any project component, at any level, and see its name and failure probability value (see Fig. 12). The information represent on two visualization corresponds to the real project file structure (see Fig. 11).

We can also navigate through displayed data by clicking on the component he wants to expand. Expanded components shows their sub-components (see Fig. 12).

If the user click on an open component, it will be collapsed. This way, user can choose what is the group of components he wants to see. When the user presses “space” key, all tree components will be displayed, the system tree visualization will be totally expanded (see Fig. 13). On complex systems (a big project with several files and lines of code) it is better to navigate in a step-by-step approach (by expanding only the desired sub-components) than by starting to see all the project components at once.

When we deal with complex systems it is difficult to have a quick notion about the tested system structure. So, when zooming into a specific area of the visualization, we lost our sense of location. By zooming in, it is not possible to see all leafs (see Fig. 14).

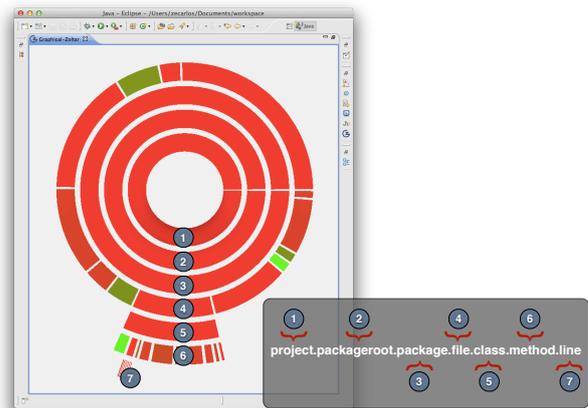


Figure 11: Levels in Sunburst and indications.

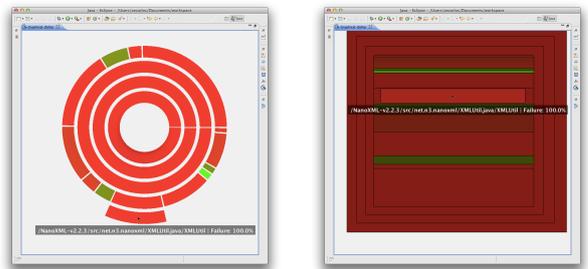


Figure 12: Navigation in GZoltar visualizations. User can expand just the needed SUT components.

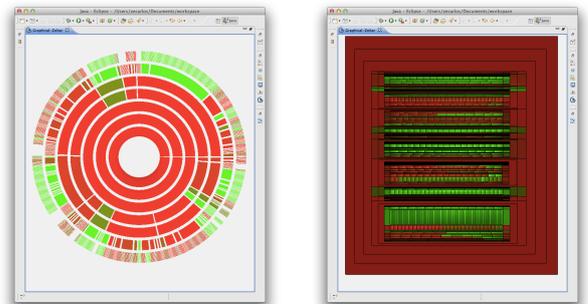


Figure 13: Expanding all components in GZoltar visualizations. User can expand all SUT components at once by pressing space key.

Root change can be seen as a “smart zoom”, because the viewing area gets limited to increase visualization detail, but maintaining the same visual structure concept. We place the mouse cursor at any element on any visualization, and click with **right-click**. Thus all the elements that are not related directly with the descendants or ascendants of the selected element, will be hidden from the visualization (see Fig. 15). This can be verified in both visualizations, Sunburst and Treemap.

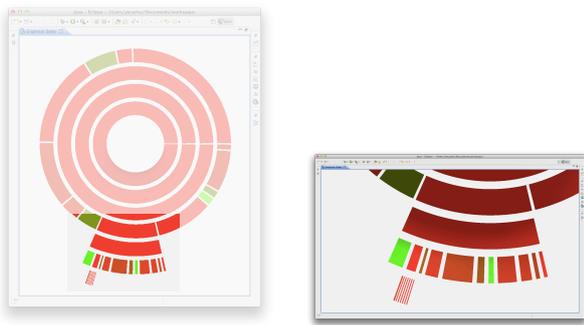


Figure 14: Zoom and Pan on NanoXML program. Detail of a program area using zoom and pan feature (right image). Left image highlights zoomed area.

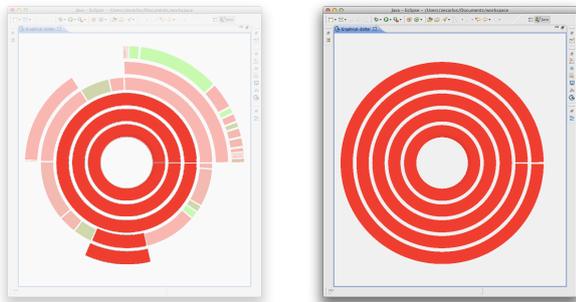


Figure 15: Root Change on NanoXML program. Detail of a program area using root change feature (right image). Left image highlights affected area.

GZOLTAR view, such RZOLTAR integrates very well with Eclipse features. GZOLTAR can open directly the standard Eclipse code editor, and position the text cursor on the wanted line (see Fig. 16). It also generates standard Eclipse warning markers that can be seen in the vertical ruler, immediately before each line of code (see Fig. 17). These markers are color-coded according to the respective line's failure probability (see Fig. 4), ranging from red (high probability of being faulty) to green (low failure probability). These markers also show tooltips with the failure probability value when the mouse is hovering them. The warning markers are also available at standard Eclipse's "Problems" View (see Fig. 18), and user can also jump to the respective class and line of code by double clicking on the warning message he wants.

After we found the line with the highest fault probability (see Fig. 19), we found the bug previously injected. We correct the bug, in other words, change the Line number 109 at XMLUtil.java file from "case ']' ':'" to "case '>>:'" (see Fig. 20).

Selecting and Re-running one of the several minimum sets provided by RZOLTAR view, we can check that everything is ok now. No warnings and no faults detected by GZOLTAR plug-in (see Fig. 21).

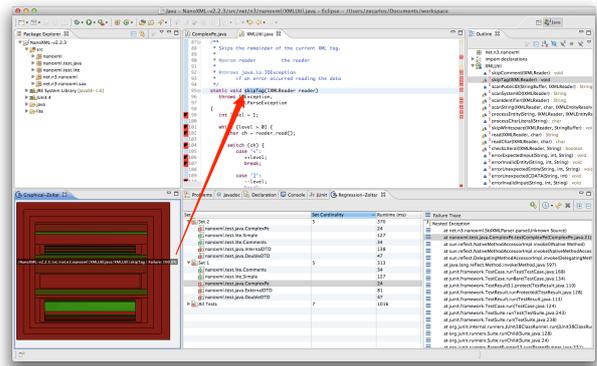


Figure 16: Code Editor Integration. When user clicks on a line of code representation, the corresponding code editor is automatically opened.

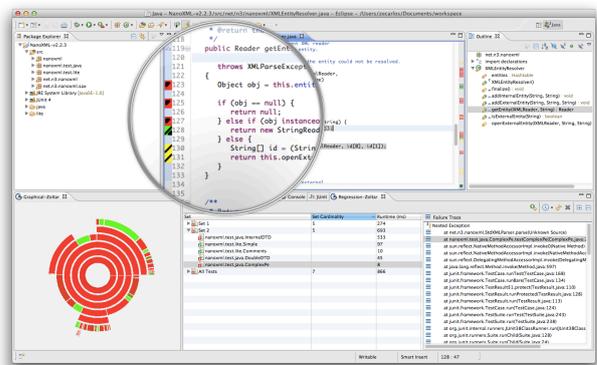


Figure 17: Warning markers next to line of code. Immediately before the line of code, the user finds markers that reveal if that line has or not any failure probability. Those signs have tooltips with the failure probability value.

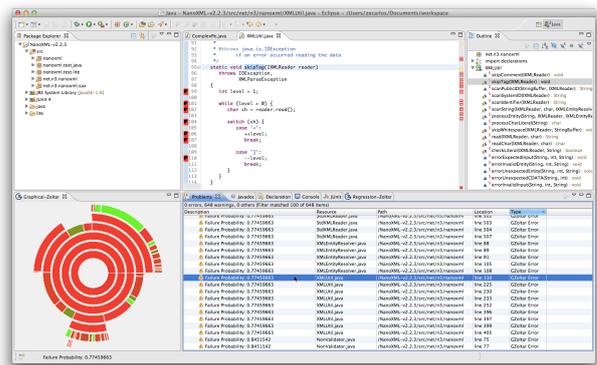


Figure 18: Problems View. Because GZoltar uses standard Eclipse warnings (but with other icon), they are also shown on Eclipse "Problems" View.

it is also compatible with 32 and 64 bit Central Processing Unit (CPU) architectures.

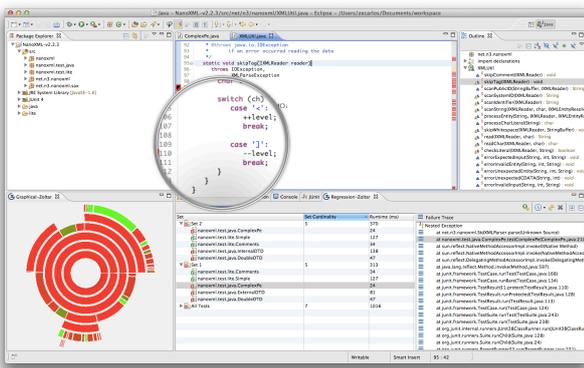


Figure 19: Line with bug.

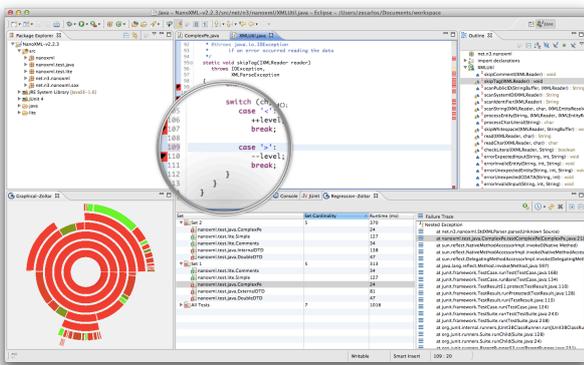


Figure 20: Line without bug.

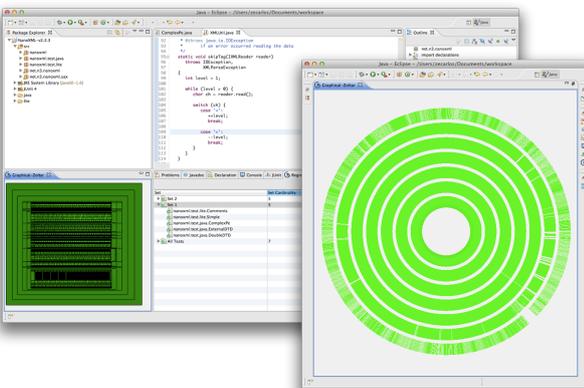


Figure 21: Everything is ok. NanoXML program without any fault.

A.2 Tool availability

Currently, the GZOLTAR plug-in is available under a request license at <http://www.gzoltar.com>. Being an Eclipse plug-in, GZoltar is installed just like any other plug-in, but have the particularity of requiring the selection of the system architecture during installation. It is compatible with Microsoft Windows, Mac OS X and also Linux Systems, and

A Dynamic Code Coverage Approach to Maximize Fault Localization Efficiency

Alexandre Perez, André Ribeiro, Rui Abreu
Department of Informatics Engineering
Faculty of Engineering, University of Porto
Porto, Portugal

{alexandre.perez, andre.riboira}@fe.up.pt, rui@computer.org

Abstract—Spectrum-based fault localization is amongst the most effective techniques for automatic fault localization. However, abstraction of program execution traces (also known as program spectra), one of the required inputs for this technique, require instrumentation of the software under test at a statement level of granularity in order to compute a list of the potential faulty statements. This introduces a considerable overhead in the fault localization process, which can even become prohibitive in, e.g., resource constrained environments. To counter this problem, we propose a new approach, coined Dynamic Code Coverage (DCC), aimed at reducing this instrumentation overhead. This technique, by means of using coarser instrumentation, starts by analyzing coverage traces for large components of the system under test. It then progressively increases the instrumentation detail for faulty components, until the statement level of detail is reached. To assess the validity of our proposed approach, an empirical evaluation was performed, injecting faults in four real-world software projects. The empirical evaluation demonstrates that the dynamic code coverage approach reduces the execution overhead that exists in spectrum-based fault localization, and even presents a more concise potential fault ranking to the user. We have observed execution time reductions of 27% on average and diagnostic report size reductions of 63% on average.

Keywords—Debugging; dynamic code coverage; software instrumentation; spectrum-based fault localization.

I. INTRODUCTION

Automatic fault localization techniques aid developers/testers to pinpoint the root cause of software failures, thereby reducing the debugging effort. Amongst the most diagnostic effective techniques is spectrum-based fault localization (SFL). SFL is a statistical technique that uses abstraction of program traces (also known as program spectra) to correlate software component (e.g., statements, methods, classes) activity with program failures [1], [2], [3]. As SFL is typically used to aid developers in identifying the root cause of observed failures, it is used with low-level of granularity (*i.e.*, statement level).

Statistical approaches are very attractive because of the relatively small overhead with respect to CPU time and memory requirement [1], [4]. However, gathering the input information, per test case, to yield the diagnostic ranking may still impose a considerable (CPU time) overhead. This is particularly the case for resource constrained environment, such as embedded systems.

As said before, typically, SFL is used at development-time at a statement level granularity (since debugging requires to locate the faulty statement). But not all components need to be inspected at such fine grain granularity. In fact, components that are unlikely to be faulty do not need to be inspected. With this reasoning in mind, we propose a technique, coined Dynamic Code Coverage (DCC), that automatically adjusts the granularity per component. First, our approach instruments the source code using a coarse granularity (e.g., package level in Java), and then decides which components to *expand* based on the output of the fault localization technique. With expanding we mean changing the granularity of the instrumentation (e.g., in Java, for instance, instrument classes, then methods, and finally statements). This expansion can be done in different ways, either selecting the components whose fault coefficient is above a certain threshold, or selecting the first ranked components, according to a set percentage.

Our empirical evaluation demonstrates that DCC has the potential to reduce drastically the execution overhead, while still maintaining the diagnostic effectiveness of statement-based spectrum-based fault localization. In our experiments, we have observed a time reduction of 27% on average. Furthermore, the rankings are easier to understand because less components are presented to the user, as low probability components are not expanded. A 63% reduction of the diagnostic report size was observed in our empirical evaluation.

In particular, this paper makes the following main contributions:

- DCC, a technique that automatically decides the instrumentation granularity for each module in the system, has been proposed; and
- An empirical study to validate the proposed technique, demonstrating its efficiency using real-world, large programs. The empirical results shows that DCC can indeed decrease the overhead imposed in the software under test, while still maintaining the same diagnostic accuracy as current approaches to fault localization. DCC also decreases the diagnostic report size when compared to traditional SFL.

To the best of our knowledge, our dynamic code coverage approach has not been described before. The remainder of

this paper is organized as follows. In the next section we present concepts relevant to this paper as well as a motivational example for our work. In Section III the dynamic code coverage approach, DCC, is described. In Section IV the findings of our empirical evaluation are presented. We compare DCC with related work in Section V. In Section VI we conclude and discuss future work.

II. CONCEPTS & MOTIVATIONAL EXAMPLE

In this section, we introduce the concept of program spectra, and its use in fault localization. Throughout this paper, we use the following terminology [5]:

- 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.

In this paper, 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.

Definition 1 A software program Π is formed by a sequence of one or more M statements.

Given its dynamic nature, central to the fault localization technique considered in this paper is the existence of a test suite.

Definition 2 A test suite $T = \{t_1, \dots, t_N\}$ 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 $|T| = N$.

Definition 3 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.

A. Program Spectra

A program spectra is a characterization of a program's execution on a dataset [6]. This collection of data, gathered at runtime, provides a view on the dynamic behavior of a program. The data consists of counters or flags for each software component. Various different program spectra exist [7], such as path-hit spectra, data-dependence-hit spectra, and block-hit spectra.

In order to obtain information about which components were covered in each execution, the program's source code needs to be instrumented, similarly to code coverage tools [8]. This instrumentation will monitor each component and register those that were executed. Components can be of several detail granularities, such as classes, methods, and lines of code.

B. Fault Localization

A fault localization technique that uses program spectra, called Spectrum-based Fault Localization (SFL), exploits information from passed and failed system runs. A passed run is a program execution that is completed correctly, and a failed run is an execution where an error was detected [1]. The criteria for determining if a run has passed or failed can be from a variety of different sources, namely test case results and program assertions, among others. The information gathered from these runs is their hit spectra [9].

The hit spectra of N runs constitutes a binary $N \times M$ matrix A , where M corresponds to the instrumented components of the program. Information of passed and failed runs is gathered in a N -length vector e , called the error vector. The pair (A, e) serves as input for the SFL technique, as seen in Figure 1.

$$\begin{array}{ccc}
 & M \text{ components} & \text{error} \\
 & & \text{detection} \\
 N \text{ spectra} & \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1M} \\ a_{21} & a_{22} & \cdots & a_{2M} \\ \vdots & \vdots & \ddots & \vdots \\ a_{N1} & a_{N2} & \cdots & a_{NM} \end{bmatrix} & \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_N \end{bmatrix}
 \end{array}$$

Figure 1: Input to SFL.

With this input, fault localization consists in identifying what columns of the matrix A resemble the vector e the most. For that, several different similarity coefficients can be used [10]. One of the most effective is the Ochiai coefficient [11], used in the molecular biology domain:

$$s_O(j) = \frac{n_{11}(j)}{\sqrt{(n_{11}(j) + n_{01}(j)) \times (n_{11}(j) + n_{10}(j))}} \quad (1)$$

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

$$n_{pq}(j) = |\{i \mid a_{ij} = p \wedge e_i = q\}| \quad (2)$$

SFL can be used with program spectra of several different granularities. However, it is most commonly used at the line of code (LOC) level and at the basic block level. Using coarser granularities would be difficult for programmers to investigate if a given fault hypothesis generated by SFL was, in fact, faulty. Throughout this work, we will be using a LOC level as the instrumentation granularity for the fault localization diagnosis report.

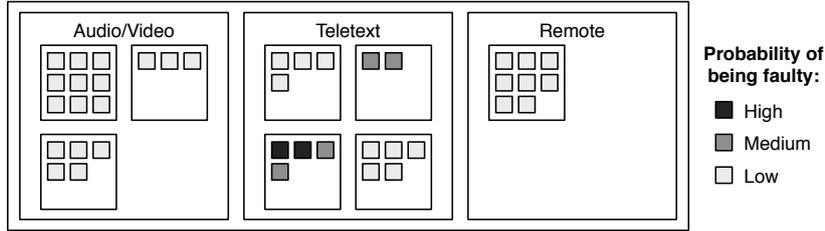


Figure 2: SFL output example.

C. Motivational Example

Suppose a program responsible for controlling a television set is being debugged. Consider that such program has three main high-level modules:

- 1) Audio and video processing;
- 2) Teletext decoding and navigation;
- 3) Remote-control input.

If one is to use SFL to pinpoint the root cause of observed failures, hit spectra for the entire application have to be gathered. Furthermore, the hit spectra have to be of a fine granularity, such as LOC level, so that the fault is more easily located.

An output of the SFL technique applied to this specific example can be seen in Figure 2. The smaller squares represent each LOC of the program, which are grouped into methods, and then into the three main modules of the program under test.

As seen in Figure 2, every LOC in the program has an associated fault coefficient that represents the probability of that component being faulty. In this example, the bottom-left function of the teletext decoding and navigation module has two LOCs with high probability of being faulty, and other two with medium probability. The upper-right function of the teletext module also contains two medium probability LOCs. There are, however, many LOCs with low probability of containing a fault. In fact, in some methods, and even entire modules, such as the audio/video processing and remote-control modules, all components have low probability. Such low probability is an indication that the fault might be located elsewhere, and thus these components need not to be inspected first.

As SFL needs to have information about the entire program spectra to perform an analysis on the most probable fault locations, this can lead to scalability problems, as every LOC has to be instrumented. Instrumentation can hit execution time by as much as 50% in code coverage tools that use similar instrumentation techniques [8]. As such, fault localization that uses hit spectra is acceptable for debugging software applications, but may be impractical for large, real-world, and resource-constrained projects that contain hundreds of thousands of LOCs.

In order to make SFL amenable to large, real, and resource-constrained applications, a way to avoid instrumenting the entire program must be devised, while still having a fine granularity for the most probable locations in the fault localization results.

III. DYNAMIC CODE COVERAGE

In order to solve the potential scaling problem that automated fault localization tools have, we propose a dynamic approach, called DCC. This method uses, at first, a coarser granularity level of instrumentation for the initial program spectra gathering. After that, it progressively increases the instrumentation detail of potential faulty components.

In Algorithm 1 it is shown the Dynamic Code Coverage algorithm. It takes as parameters *System* and *TestSuite*, corresponding to the System Under Test (SUT) and its test suite, respectively.

Algorithm 1 Dynamic Code Coverage.

```

1: procedure DCC(System, TestSuite)
2:    $\mathcal{R} \leftarrow \emptyset$ 
3:    $\mathcal{F} \leftarrow \text{System}$ 
4:    $\mathcal{T} \leftarrow \text{TestSuite}$ 
5:    $\mathcal{G} \leftarrow \text{INITIALGRANULARITY}$ 
6:   repeat
7:     INSTRUMENT( $\mathcal{F}$ ,  $\mathcal{G}$ )
8:     ( $A$ ,  $e$ )  $\leftarrow$  RUNTESTS( $\mathcal{T}$ )
9:      $\mathcal{C} \leftarrow \text{SFL}(A, e)$ 
10:     $\mathcal{F} \leftarrow \text{FILTER}(\mathcal{C})$ 
11:     $\mathcal{R} \leftarrow \text{UPDATE}(\mathcal{R}, \mathcal{F})$ 
12:     $\mathcal{T} \leftarrow \text{NEXTTESTS}(\text{TestSuite}, A, \mathcal{F})$ 
13:     $\mathcal{G} \leftarrow \text{NEXTGRANULARITY}(\mathcal{F})$ 
14:  until ISFINALGRANULARITY( $\mathcal{F}$ )
15:  return  $\mathcal{R}$ 
16: end procedure

```

First, an empty report \mathcal{R} is created. After that, a list of the components to instrument \mathcal{F} is initialized with all *System* components. Similarly, the list of test cases to run in each iteration \mathcal{T} is initialized with all test cases in *TestSuite*. An initial granularity \mathcal{G} is also calculated with the method

INITIALGRANULARITY, which can be set from a class level to a LOC level.

After the initial assignments, the algorithm will start its iteration phase in line 6. At the start of each iteration, every component in the list \mathcal{F} is instrumented with the granularity \mathcal{G} with the method INSTRUMENT. What this method does is to alter these components so that their execution is registered in the program spectra.

Afterwards, the test cases \mathcal{T} are run with the method RUNTESTS. Its output is a hit spectra matrix A for all the previously instrumented components, and the error vector e , that states what tests passed and what tests failed. As explained in Section II-B, these are the necessary inputs for spectrum-based fault localization, performed in line 9. This SFL method calculates, for each instrumented component, its failure coefficient using the Ochiai coefficient, shown in equation 1.

Following the fault localization step, the components are passed through a FILTER that eliminates the low probability ones according to a set threshold, and the list \mathcal{F} is updated, as well as the fault localization report \mathcal{R} .

In line 12, the test case set is updated to run only the tests that touch the current components \mathcal{F} . Such tests can be retrieved by analyzing the coverage matrix A .

The last step in the iteration is to update the instrumentation granularity for next iterations. Method NEXTGRANULARITY finds the coarser granularity in all the components of list \mathcal{F} , and updates that granularity to the next level of detail.

Every iteration is tested for recursion with ISFINALGRANULARITY, that returns true if every component in the list \mathcal{F} is at the desired final granularity, such as LOC or basic bloc granularities. Lastly, the DCC algorithm returns the fault localization report \mathcal{R} .

DCC's performance is very dependent on the FILTER function, which is responsible to decide whether or not it is required to zoom-in¹ in a given component. Although many filters may be plugged into the algorithm, in this paper we study the impact of two filters (see Figure 3 for an illustration):

- Coefficient filter C_f – components above the SFL coefficient threshold C_f are expanded.
- Percentage filter P_f – the first $P_f\%$ components are expanded.

The main advantages of our dynamic code coverage algorithm, DCC, are twofold. The first one is the decrease of instrumentation overhead in the program execution (as demonstrated by the empirical results). This is due to the fact that not every LOC is instrumented – only the LOCs most likely to contain a fault will be instrumented at that level of detail.

¹In this context, zooming-in is to explore the inner components of a given component.

Coefficient Filter (> 0.6)					
0.9	0.8	0.8	0.7	0.5	0.1
0.7	0.2	0.2	0.1	0.0	0.0

Percentage Filter (50%)					
0.9	0.8	0.8	0.7	0.5	0.1
0.7	0.2	0.2	0.1	0.0	0.0

Figure 3: Component filters.

The second advantage is the fact that, in every iteration, the generated program spectra matrices, seen in line 8 of Algorithm 1, will be shorter in size when compared to traditional SFL. That way, the fault coefficient calculation, described in Section II-B, will be inherently faster, as there are fewer components to calculate.

The iterative nature of the DCC algorithm also provides some benefits. In each iteration, the algorithm is walking towards a solution, narrowing down the list of components which are likely to contain a fault. As such some information about those components can be made available, directing the developer to the fault location even before the algorithm is finished. Secondly, as low probability components are being filtered, the final report will also be shorter, providing the developer with a more concise fault localization report.

To illustrate the overhead reduction, let us revisit the motivational example given in Section II-C. If use the DCC approach to debug this program, we get the output shown in Figure 4. In this example, a filter responsible for not exploring components with low probability of containing faults is being used. In particular, the algorithm executes as follows:

- 1) The three modules – Audio/Video, Teletext, and Remote – are instrumented at the module level. Upon running the tests and SFL, the only component with high probability of being faulty is the Teletext module.
- 2) The Teletext module is instrumented at a method level. After that, the tests that touch the Teletext module are run. Fault localization states that the upper-right (UR) and the bottom-left (BL) functions have medium and high probability of being faulty, respectively.
- 3) The UR and BL functions are instrumented at the LOC level. After the tests that touch those functions are run and fault localization is performed, every LOC in those functions has an associated fault coefficient. As all the non-filtered components are of LOC granularity, the execution is terminated.

This approach, besides only reporting LOCs which are more likely to contain a fault, also needed to instrument less software components – 13 in total. Compared to the

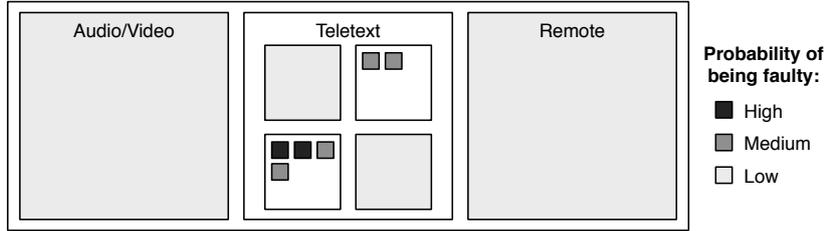


Figure 4: DCC output example.

pure SFL approach of Section II-C, where 40 components were instrumented, DCC has reduced instrumentation (thus, overhead) by 67.5%.

IV. EMPIRICAL EVALUATION

In this section, we evaluate the validity and performance of the DCC approach for real projects. First, we introduce the programs under analysis and the evaluation metrics. Then, we discuss the empirical results and finish this section with a threats to validity discussion.

A. Experimental Setup

For our empirical study, four subjects written in Java were considered:

- NanoXML² – a small XML parser.
- `org.jacoco.report` – report generation module for the JaCoCo³ code coverage library.
- XML-Security – a component library implementing XML signature and encryption standards. This library is part of the Apache Santuario⁴ project.
- JMeter⁵ – a desktop application designed to load test functional behavior and measure performance of web applications.

The project details of each subject are in Table I. The LOC count information was gathered using the metrics calculation and dependency analyzer plugin for Eclipse Metrics⁶. Test count and coverage percentage were collected with the Java code coverage plugin for Eclipse EclEmma⁷.

Subject	Version	LOCs (M)	Test Cases	Coverage
NanoXML	2.2.6	5393	8	53.2%
<code>org.jacoco.report</code>	0.5.5	5979	225	97.2%
XML-Security	1.5.0	60946	461	59.8%
JMeter	2.6	127359	593	34.2%

Table I: Experimental Subjects.

To assess the efficiency and effectiveness of DCC the following experiments were performed, using fifteen faulty

²NanoXML – <http://devkix.com/nanoxml.php>

³JaCoCo – <http://www.eclemma.org/jacoco/index.html>

⁴Apache Santuario – <http://santuario.apache.org/>

⁵JMeter – <http://jmeter.apache.org/>

⁶Metrics – <http://metrics.sourceforge.net/>

⁷EclEmma – <http://www.eclemma.org/>

versions per subject program. We injected one fault in each of the 15 versions⁸:

- SFL without DCC. This is the reference baseline.
- DCC with constant value coefficient filters from 0 to 0.95, with intervals of 0.05.
- DCC with percentage filters from 100% to 5%, with intervals of 5%.

The metrics gathered were the fault localization execution time, the size of the fault localization report, and the average LOCs needed to be inspected until the fault is located. The latter metric can be calculated by sorting the fault localization report by the value of the coefficient, and finding the injected fault’s position. In this metric, we are assuming that the developer performs the inspection in an ordered manner, starting from the highest fault coefficient LOCs.

As spectrum-based fault localization creates a ranking of components in order of likelihood to be at fault, we can retrieve how many components we still need to inspect until we hit the faulty one. Let $d \in \{1, \dots, K\}$, where K is the number of ranked components and $K \leq M$, be the index of the statement that we know to contain the fault. For all $j \in \{1, \dots, M\}$, let s_j . Then the ranking position of the faulty statement is given by

$$\tau = \frac{|\{j|s_j > s_d\}| + |\{j|s_j \geq s_d\}| - 1}{2} \quad (3)$$

$|\{j|s_j > s_d\}|$ counts the number of components that outrank the faulty one, and $|\{j|s_j \geq s_d\}|$ counts the number of components that rank with the same probability as the faulty one plus the ones that outrank it.

We define quality of diagnosis as the effectiveness to pinpoint the faulty component. As said before, this metric represents the percentage of components that need not be considered when searching for the fault by traversing the ranking. It is defined as

$$\left(1 - \frac{\tau}{K_{SFL}}\right) \cdot 100\% \quad (4)$$

where K_{SFL} is the number of ranked components of SFL without DCC – the reference baseline.

⁸In the future, we plan to assess the effectiveness of DCC when tackling multiple bugs, by injecting several faults at once.

The experiments were run on a 2.7 GHz Intel Core i7 MacBook Pro with 4 GB of RAM, running OSX Lion.

B. Experimental Results

Figures 5, 6, 7 and 8 summarize the overall execution time outcomes for all the experimental subjects. Each figure contains two plots, detailing the fault localization execution of each injected fault with DCC using constant coefficient value filters and with DCC using percentage filters, respectively. These filtering methods were previously detailed in Section III. Please note that these results are gathered by running the entire fault localization experiments detailed in the previous section, and do not pertain only to the instrumentation overhead.

Due to space constraints, only three thresholds are shown for both filters: 0.0, 0.25 and 0.5 thresholds for the constant coefficient value filters (C_f) and 50%, 30% and 10% thresholds for the percentage filters (P_f). To obtain a better understanding of the performance of each experiment, we also added, for every injected fault, the fault localization execution time of the SFL without DCC approach, labeled “No DCC” in the aforementioned figures. This way, DCC approaches can be easily compared with the SFL approach. Unless stated otherwise, every fault localization execution is able to find the injected fault (*i.e.*, the resulting report contains the injected fault).

The first experimental subject to be analyzed was the NanoXML project, whose experiment results can be seen in Figure 5. Note that experiments 01, 11 and 13 for $C_f = 0.5$ (see Figure 5a) and the experiments 01, 04, 11, 12, 13, 14, 15 for $P_f = 10\%$ (see Figure 5b) were not able to find the injected faults.

As we can see from the experiment results, the DCC approach underperforms the current SFL method based in the execution time. Such results can be explained if we analyze the NanoXML project information in Table I. This project, not only is rather small in size, but also has very few test cases. At the same time, it has a coverage of over 50%. What this means is that some test cases, if not all, touch many different statements. As such, the generated program spectra matrices, detailed in Sections II-A and II-B will be rather dense. Because of this, many components will have similar coefficients, rendering the filtering operation ineffective: either discarding many different components, or keeping a lot of components to be re-instrumented and re-tested.

The next analyzed subject was `org.jacoco.report`, part of the JaCoCo project. The filters $C_f = 0.5$ (see Figure 6a) and $P_f = 10\%$ (see Figure 6b) were both not able to find the injected faults in experiments 09 and 15. Also, the injected fault in experiment 02 was not found in $P_f = 10\%$.

This subject, despite having many more test cases than the previous project, still has some performance drops in some

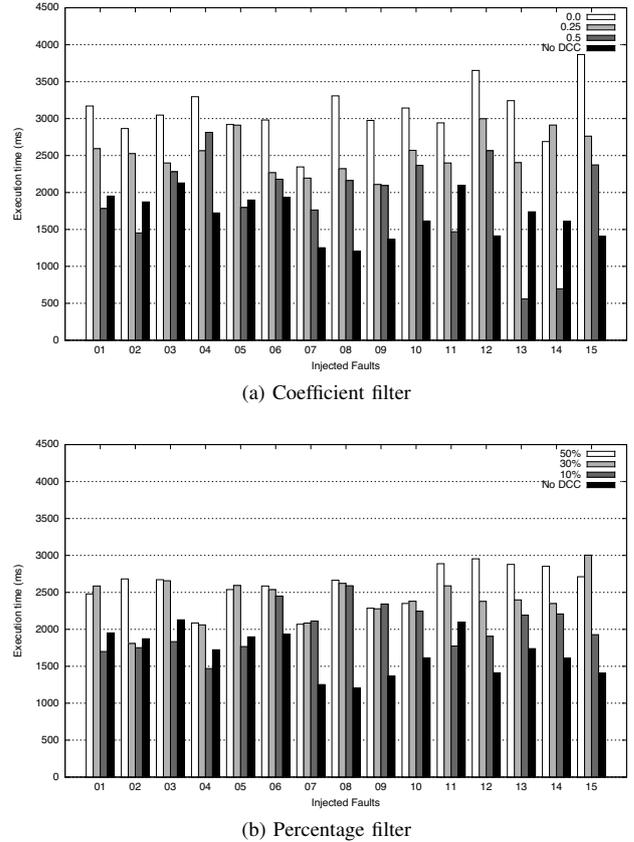


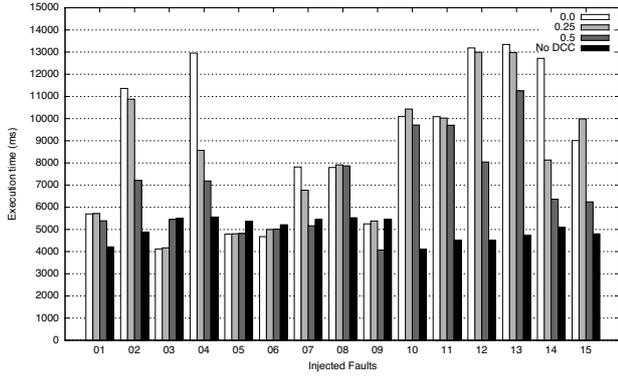
Figure 5: NanoXML time execution results.

of the experiments. Upon investigating the fault localization reports of the lower performance experiments, we realized that their length can be as high as 950 statements in some experiments. This means that the set of test cases that touch the injected faulty statements can cover roughly 15% of the entire project. Because of this, the same thing as the previous project happens: many components will have similar coefficients, rendering the expansions ineffective.

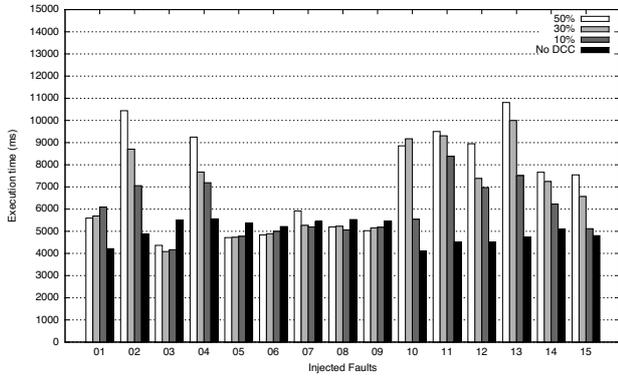
The following subject was the XML-Security project. Injected faults in experiments 03 and 08 were not found by $C_f = 0.5$ (see Figure 7a). Experiment 08 also did not have its injected fault in $P_f = 10\%$ (see Figure 7b).

The last subject was the JMeter project. Injected faults were not found by $C_f = 0.5$ (see Figure 8a) in experiments 01 and 11. Every fault was found with the percentage filters.

Both XML-Security and JMeter have better results when utilizing DCC. There are mainly two reasons for these results. The first is the fact that the program spectra matrix is sparser. The other important reason is, as programs grow in size, the overhead of a fine-grained instrumentation, used in methodologies such as SFL, is much more noticeable. In this kind of sizable projects (see project informations in Table I), and if the matrix is sparse enough, it is preferable



(a) Coefficient filter



(b) Percentage filter

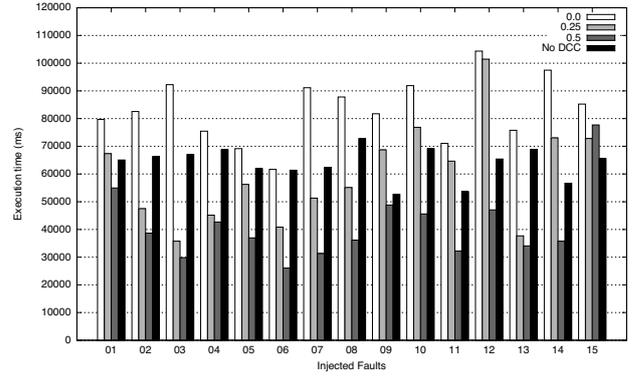
Figure 6: org.jacoco.report time execution results.

to re-run some of the tests, than to instrument every LOC at the start of the fault localization process.

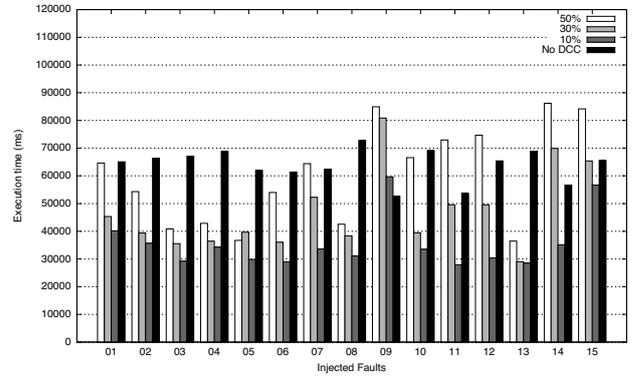
These time execution results confirm our assumptions that DCC can over-perform SFL for larger projects, where the instrumentation overhead is heavily noticeable. In contrast, for smaller projects, DCC does suffer in performance, mainly due to the fact that the overhead of re-running tests produces a bigger performance hit than the instrumentation granularity overhead. In fact, if we take into account all experiments for all four projects, there actually is an increase of execution time of 8% ($\sigma = 0.48$)⁹. However, if we only consider the larger projects where instrumentation is a more prevalent issue (*i.e.* XML-Security and JMeter), the dynamic code coverage approach can reduce execution time by 27% on average ($\sigma = 0.28$).

The other gathered metrics in this empirical evaluation, unlike execution time, show a consistent improvement over SFL in every project. In average, the DCC approach reduced 63% ($\sigma = 0.30$) the generated fault localization ranking, providing a more concise report when compared to SFL. The

⁹We have chosen to use the metrics gathered by the $P_f = 30\%$ filter since it is the best performing filter of those considered in this section that is able to find the injected faults for every experiment.



(a) Coefficient filter



(b) Percentage filter

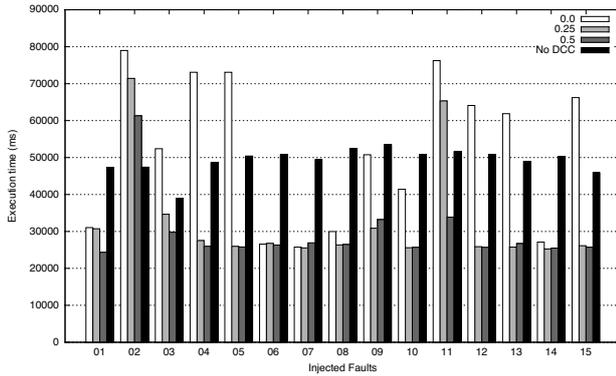
Figure 7: XML-Security time execution results.

quality of diagnosis, described in equation 4, also suffered a slight improvement, from 85% ($\sigma = 0.20$) without DCC to 87% ($\sigma = 0.19$) with DCC.

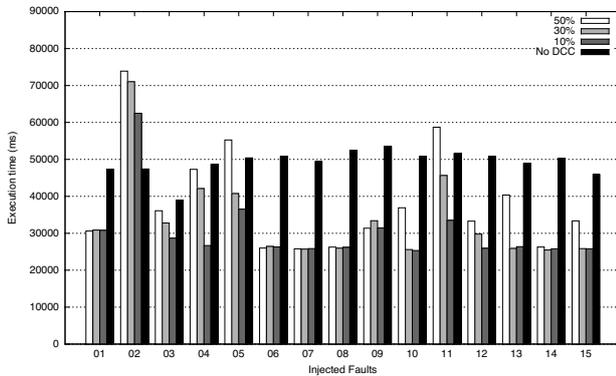
C. Threats to Validity

The main threat to external validity of these empirical results is the fact that only four test subjects were considered. Although the subjects were all real, open source software projects, it is plausible to assume that a different set of subjects, having inherently different characteristics, may yield different results. Other threat to external validity is related to the injected faults used in the experiments. These injected faults, despite being fifteen in total for each experimental subject, may not represent the entire conceivable software fault spectrum.

Threats to internal validity are related to some fault in the DCC implementation, or any underlying implementation, such as SFL or even the instrumentation for gathering program spectra. To minimize this risk, some testing and individual result checking were performed before the experimental phase.



(a) Coefficient filter



(b) Percentage filter

Figure 8: JMeter time execution results.

V. RELATED WORK

The process of pinpointing the fault(s) that led to symptoms (failures/errors) is called fault localization, and has been an active area of research for the past decades. Based on a set of observations, automatic approaches to software fault localization yield a list of likely fault locations, which is subsequently used either by the developer to focus the software debugging process. Depending on the amount of knowledge that is required about the system’s internal component structure and behavior, the most predominant approaches can be classified as (1) statistical approaches or (2) reasoning approaches. The former approach uses an abstraction of program traces, dynamically collected at runtime, to produce a list of likely candidates to be at fault, whereas the latter combines a static *model* of the expected behavior with a set of observations to compute the diagnostic report.

Statistics-based fault localization techniques, as stated above, use an abstraction of program traces, also known as *program spectra*, to find a statistical relationship with observed failures. Program spectra are collected at run-time, during the execution of the program, and many different forms exist [7]. For example, component-hit spectra indicate

whether a component was involved in the execution of the program or not. In contrast to model-based approaches, program spectra and pass/fail information are the only *dynamic* source of information used by statistics-based techniques.

Well-known examples of such approaches are the Tarantula tool by Jones, Harrold, and Stasko [12], the Nearest Neighbor technique by Renieris and Reiss [13], the Sober tool by Lui, Yan, Fei, Han, and Midkiff [2], the work of Liu and Hand [14], CrossTab by Wong, Wei, Qi, and Zap [3], the Cooperative Bug Isolation (CBI) by Liblit and his colleagues [15], [16], [17], [18], the Time Will Tell approach by Yilmaz, Paradkar, and Williams [19], HOLMES by Chilimbi *et al.* [20], and MKBC by Xu, Chan, Zhang, Tse, and Li [21]. Although differing in the way they derive the statistical fault ranking, all techniques are based on measuring program spectra. Note that this list is by no means exhaustive.

Toolsets providing fault localization using spectrum-based fault localization exist, namely in Zoltar [22] and Tarantula [23], [12] for C projects, and GZoltar [24] for Java projects. However, none of these tools employ a dynamic code coverage approach to SFL, having to instrument the entire SUT. Also, their instrumentation granularity is set at a LOC level of detail. A DCC approach could certainly be added to any of these tools, with minimal algorithmic changes, provided that the underlying instrumentation tool that these tools use to gather program spectra supports different levels of detail.

Reasoning approaches to fault localization use prior knowledge of the system, such as required component behavior and interconnection, to build a model of the correct behavior of the system. An example of a reasoning technique is model-based diagnosis (see, e.g., [25]), where a diagnosis is obtained by logical inference from the *static* model of the system, combined with a set of run-time observations. In the software engineering community this approach is often called model-based software debugging [26]. Well-known approaches to model-based software debugging include the approaches of Friedrich, Stumptner, and Wotawa [27], [28], Nica and Wotawa [29], Wotawa, Stumptner, and Mayer [30], and Mayer and Stumptner [26].

As model-based techniques technique may suffer from large diagnostic results and not scale to sizable projects, some work was already combining SFL with Model-Based Software Debugging (MBSD) has been proposed [31], [32], where MBSD is used to refine the output report generated by the spectrum-based fault localization, filtering the components that do not explain the observed failures. Our DCC approach could also be combined with this technique, by performing the fault localization until a certain middle-grained level of component detail (*e.g.*, method level), and submit the top components to be analyzed by MBSD.

VI. CONCLUSIONS & FUTURE WORK

We have shown that current approaches to spectrum-based fault localization face some challenges concerning scalability due to the input gathering overhead caused by a fine grained instrumentation throughout the system under test. For instance, this may be an issue in resource-constrained systems. A solution to this problem was presented, coined Dynamic Code Coverage (DCC), that initially uses a coarser granularity of instrumentation, and progressively increases the instrumentation detail of potential faulty components. In our empirical evaluation, we have validated our approach, and demonstrated that it not only reduces the execution time by 27% on average, but also reduces the number of components reported to the user by 63% on average.

As for future work, some aspects of the dynamic code coverage technique still require further investigation. One of those is the way of how the initial system granularity is established. Currently, this value is set manually and is the same across the entire system under test. A way to change this would be by using static analysis to assess program information and to adjust the system's initial granularity accordingly. Another approach would be to learn what were the most frequently expanded components from previous executions, and change these components' initial granularity independently. Other issue that requires further investigation pertains to the filtering methods. It is possible that there are better filtering methods than the ones presented in this paper, namely methods that employ dynamic strategies, that change the cutting threshold based on program spectra analysis.

REFERENCES

- [1] R. Abreu, P. Zoetewij, R. Golsteijn, and A. J. C. Van Gemund, "A practical evaluation of spectrum-based fault localization," *Journal of Systems and Software*, vol. 82, no. 11, pp. 1780–1792, 2009.
- [2] C. Liu, L. Fei, X. Yan, J. Han, and S. Midkiff, "Statistical debugging: A hypothesis testing-based approach," *IEEE Transactions on Software Engineering (TSE)*, vol. 32, no. 10, pp. 831–848, 2006.
- [3] E. Wong, T. Wei, Y. Qi, and L. Zhao, "A crosstab-based statistical method for effective fault localization," in *Proceedings of the 1st International Conference on Software Testing, Verification, and Validation (ICST'08)*, R. Hierons and A. Mathur, Eds. Lillehammer, Norway: IEEE Computer Society, 9 – 11 April 2008, pp. 42–51.
- [4] R. Abreu, P. Zoetewij, and A. J. C. van Gemund, "Spectrum-based multiple fault localization," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE'09)*, G. Taentzer and M. Heimdahl, Eds. Auckland, New Zealand: IEEE Computer Society, to appear, 16 – 20 November 2009.
- [5] A. Avižienis, J.-C. Laprie, B. Randell, and C. E. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable Secure Computing*, vol. 1, no. 1, pp. 11–33, 2004.
- [6] T. Reps, T. Ball, M. Das, and J. Larus, "The use of program profiling for software maintenance with applications to the year 2000 problem," in *Proceedings of the 6th European Software Engineering conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of software engineering*, ser. ESEC '97/FSE-5. New York, NY, USA: Springer-Verlag New York, Inc., 1997, pp. 432–449.
- [7] M. J. Harrold, G. Rothermel, K. Sayre, R. Wu, and L. Yi, "An empirical investigation of the relationship between fault-revealing test behavior and differences in program spectra," *STVR Journal of Software Testing, Verification, and Reliability*, no. 3, pp. 171–194, September 2000.
- [8] Q. Yang, J. J. Li, and D. Weiss, "A survey of coverage based testing tools," in *Proceedings of the 2006 international workshop on Automation of software test*, ser. AST '06. New York, NY, USA: ACM, 2006, pp. 99–103.
- [9] R. Abreu, P. Zoetewij, and A. J. van Gemund, "An evaluation of similarity coefficients for software fault localization," in *Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing*, ser. PRDC '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 39–46.
- [10] A. K. Jain and R. C. Dubes, *Algorithms for clustering data*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1988.
- [11] R. Abreu, P. Zoetewij, and A. J. C. van Gemund, "On the accuracy of spectrum-based fault localization," in *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 89–98.
- [12] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ser. ASE '05. New York, NY, USA: ACM, 2005, pp. 273–282.
- [13] M. Renieris and S. P. Reiss, "Fault localization with nearest neighbor queries," in *Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE'03)*, J. Grundy and J. Penix, Eds. Montreal, Canada: IEEE Computer Society, 6 – 10 October 2003, pp. 30–39.
- [14] C. Liu and J. Han, "Failure proximity: a fault localization-based approach," in *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT FSE'06)*, M. Young and P. T. Devanbu, Eds. Portland, Oregon, USA: ACM Press, 5 – 11 November 2006, pp. 46–56.
- [15] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," in *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI'05)*, V. Sarkar and M. W. Hall, Eds. Chicago, Illinois, USA: ACM Press, 12 – 15 June 2005, pp. 15–26.
- [16] B. Liblit, "Cooperative debugging with five hundred million test cases," in *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'08)*, B. G. Ryder and A. Zeller, Eds. Seattle, Washington, USA: ACM Press, 20 – 24 July 2008, pp. 119–120.

- [17] P. A. Nainar, T. Chen, J. Rosin, and B. Liblit, "Statistical debugging using compound boolean predicates," in *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'07)*, D. S. Rosenblum and S. G. Elbaum, Eds. London, UK, July: ACM Press, 9 – 12 July 2007, pp. 5–15.
- [18] A. X. Zheng, M. I. Jordan, B. Liblit, M. Naik, and A. Aiken, "Statistical debugging: simultaneous identification of multiple bugs," in *Proceedings of the 23rd International Conference on Machine Learning (ICML'06)*, ser. ACM International Conference Proceeding Series, W. W. Cohen and A. Moore, Eds., vol. 148. Pittsburgh, Pennsylvania, USA: ACM Press, 25 – 29 June 2006, pp. 1105–1112.
- [19] C. Yilmaz, A. M. Paradkar, and C. Williams, "Time will tell: fault localization using time spectra," in *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*, W. Schäfer, M. B. Dwyer, and V. Gruhn, Eds. Leipzig, Germany: ACM Press, 10 – 18 May 2008, pp. 81–90.
- [20] T. M. Chilimbi, B. Liblit, K. Mehra, A. V. Nori, and K. Vaswani, "Holmes: Effective statistical debugging via efficient path profiling," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 34–44.
- [21] J. Xu, W. K. Chan, Z. Zhang, T. H. Tse, and S. Li, "A dynamic fault localization technique with noise reduction for java programs," in *Proceedings of the 11th Int. Conference on Quality Software (QSIC 2011)*, 2011, pp. 11–20.
- [22] T. Janssen, R. Abreu, and A. J. van Gemund, "Zoltar: A spectrum-based fault localization tool," in *Proceedings of the 2009 ESEC/FSE workshop on Software integration and evolution @ runtime*, ser. SINTER '09. New York, NY, USA: ACM, 2009, pp. 23–30.
- [23] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *Proceedings of the 24th International Conference on Software Engineering*, ser. ICSE '02. New York, NY, USA: ACM, 2002, pp. 467–477.
- [24] A. Ribeiro, R. Abreu, and R. Rodrigues, "An OpenGL-based eclipse plug-in for visual debugging," in *Proceedings of the 1st Workshop on Developing Tools as Plug-ins*, ser. TOPI '11. New York, NY, USA: ACM, 2011, pp. 60–60.
- [25] J. de Kleer and B. C. Williams, "Diagnosing multiple faults," *Artificial Intelligence*, vol. 32, no. 1, pp. 97–130, 1987.
- [26] W. Mayer and M. Stumptner, "Model-Based Debugging State of the Art And Future Challenges," *Electronic Notes in Theoretical Computer Science*, vol. 174, no. 4, pp. 61–82, 2007.
- [27] G. Friedrich, M. Stumptner, and F. Wotawa, "Model-based diagnosis of hardware designs," *Artificial Intelligence*, vol. 111, no. 1-2, pp. 3–39, 1999.
- [28] —, "Model-based diagnosis of hardware designs," in *Proceedings of the 12th European Conference on Artificial Intelligence (ECAI'96)*, W. Wahlster, Ed. Budapest, Hungary: John Wiley and Sons, Chichester, 11–16 August 1996, pp. 491–495.
- [29] M. Nica and F. Wotawa, "From constraint representations of sequential code and program annotations to their use in debugging," in *Proceedings of the 18th European Conference on Artificial Intelligence (ECAI'08)*, ser. Frontiers in Artificial Intelligence and Applications, M. Ghallab, C. D. Spyropoulos, N. Fakotakis, and N. M. Avouris, Eds., vol. 178. Patras, Greece: IOS Press, 21–26 July 2008, pp. 797–798.
- [30] F. Wotawa, M. Stumptner, and W. Mayer, "Model-based debugging or how to diagnose programs automatically," in *Proceedings of IAE/AIE 2002*, ser. LNCS, T. Hendtlass and M. Ali, Eds., vol. 2358. Cairns, Australia: Springer-Verlag, 17 – 20 June 2002, pp. 746–757.
- [31] W. Mayer, R. Abreu, M. Stumptner, and A. J. van Gemund, "Prioritizing model-based debugging diagnostic reports," in *Proceedings of the 19th International Workshop on Principles of Diagnosis (DX'08)*, Blue Mountains, NSW, Australia, September 2008, pp. 127–134.
- [32] R. Abreu, W. Mayer, M. Stumptner, and A. J. C. van Gemund, "Refining spectrum-based fault localization rankings," in *Proceedings of the 2009 ACM symposium on Applied Computing*, ser. SAC '09. New York, NY, USA: ACM, 2009, pp. 409–414.

ABOUT THE AUTHORS

Alexandre Perez is a student and researcher at Faculty of Engineering of University of Porto, Portugal. Currently, he is finishing his master degree in Informatics and Computing Engineering. His master's thesis is being supervised by André Ribeiro and Rui Abreu.

André Ribeiro graduated in Informatics Engineering from the College of Engineering of Porto, Portugal, carrying out his graduation project at Faculty of Medicine of the University of Porto, Portugal. He received his master degree in Informatics and Computing Engineering from the Faculty of Engineering of the University of Porto, Portugal, in 2011. He was a researcher at the University of Minho, Portugal. Currently he is a Ph.D student at the Faculty of Engineering of the University of Porto, Portugal, and is a researcher at the same faculty. He is also a researcher at the HAS-Lab / INESC TEC, Portugal.

Rui Abreu graduated in Systems and Computer Engineering from University of Minho, Portugal, carrying out his graduation thesis project at Siemens S.A., Portugal. Between September 2002 and February 2003, Rui followed courses of the Software Technology Master Course at University of Utrecht, the Netherlands, as an Erasmus Exchange Student. He was an intern researcher at Philips Research Labs, the Netherlands, between October 2004 and June 2005. He received his Ph.D. degree from the Delft University of Technology, the Netherlands, in November 2009, and he is currently an assistant professor at the Faculty of Engineering of the University of Porto, Portugal. He is also a researcher at the HAS-Lab / INESC TEC, Portugal.