

SOFTWARE MAINTENANCE AT COMMIT TIME

MATHIEU NAYROLLES

A THESIS
IN
THE DEPARTMENT
OF
ELECTRICAL & COMPUTER ENGINEERING

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
CONCORDIA UNIVERSITY
MONTRÉAL, QUÉBEC, CANADA

APRIL 2018

© MATHIEU NAYROLLES, 2018

CONCORDIA UNIVERSITY
School of Graduate Studies

This is to certify that the thesis prepared

By: **Mr. Mathieu Nayrolles**

Entitled: **Software Maintenance at Commit Time**

and submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Electrical & Computer Engineering)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

_____	Chair
_____	External Examiner
_____	Examiner
_____	Examiner
_____	Examiner
Dr Wahab Hamou-Lhadj _____	Supervisor

Approved _____
Chair of Department or Graduate Program Director

_____ 20 _____

Dean

Faculty of Engineering and Computer Science

Abstract

Software Maintenance at Commit Time

Mathieu Nayrolles, Ph.D.

Concordia University, 2018

Software maintenance activities such as debugging and feature enhancement are known to be challenging and costly. Studies have shown that the cost of software maintenance can reach up to 70% of the overall cost of the software development life cycle (Health, Social and Research 2002). Much of this is attributable to several factors including the increase in software complexity, the lack of traceability between the various artifacts of the software development process, the lack of proper documentation, and the unavailability of the original developers of the systems. The large adoption of these tools by practitioners remains limited, and factors that prevent such adoption are still an open question. In this thesis, we propose to address some of the issues mentioned above by focusing on developing techniques and tools that support software maintainers at commit-time. As part of the developer's workflow, a commit marks the end of a given task or subtask as the developer is ready to version the source code. Commits are bite-sized units of work that are potentially ready to be shared with the rest of the organization. We propose three approaches named PRECINCT, BIANCA and CLEVER. PRECINCT prevents the insertion of new software clones at commit-time while BIANCA and CLEVER prevent the introduction of new defects at commit-time. BIANCA and CLEVER also propose potential fixes to identified defects in order to support the developer. We also propose JCHARMING that can reproduce on-field crashes in case commit-time approaches failed to prevent the introduction of a defect in the system. Overall, our approaches have been tested on over 400 open- and closed- sources systems with high levels of precision and recall while being scalable and non-intrusive for developers. Finally, we also propose a taxonomy of software fault that can help researchers to categorize the research in the various areas related to software maintenance.

Acknowledgments

acknowledgments

Contents

List of Figures	x
List of Tables	xii
1 Introduction	1
1.1 Thesis Contributions	2
1.2 Thesis Organization	5
1.3 Related Publications	6
2 Background	9
2.1 Definitions	9
2.2 Version control systems	10
2.3 Providers	13
2.3.1 Project Tracking Systems	13
2.4 Providers	16
3 Related work	17
3.1 Clone Detection	17
3.1.1 Traditional Clone Detection Techniques	17
3.1.2 Remote Detection of Clones	18
3.1.3 Local Detection of Clones	19
3.2 Reports and source code relationships	21
3.3 Fault Prediction	22
3.4 Automatic Patch Generation	23
3.5 Crash Reproduction	24
3.5.1 On-field Record and In-house Replay	25

3.5.2	On-house Crash Explanation	27
3.6	Bugs Classification	29
4	An Aggregated Bug Repository for Developers and Researchers	32
4.1	Introduction	32
4.2	Approach	33
4.2.1	Architecture	33
4.2.2	BUMPER Metadata	34
4.2.3	Bumper Query Language and API	36
4.2.4	Bumper Data Repository	37
4.3	Experimental Setup	38
4.4	Empirical Validation	40
4.5	Threats to Validity	41
4.6	Chapter Summary	41
5	Preventing Code Clone Insertion At Commit-Time	43
5.1	Introduction	43
5.2	Approach	44
5.2.1	Commit	45
5.2.2	Pre-Commit Hook	46
5.2.3	Extract and Save Blocks	46
5.2.4	Compare Extracted Blocks	50
5.2.5	Output and Decision	51
5.3	Experimental Setup	53
5.4	Empirical Validation	55
5.5	Threats to Validity	59
5.6	Chapter Summary	60
6	Preventing Bug Insertion Using Clone Detection At Commit-Time	61
6.1	Introduction	61
6.2	Approach	63
6.2.1	Clustering Project Repositories	64
6.2.2	Building a Database of Code Blocks of Defect-Commits and Fix-Commits	65
6.2.3	Analysing New Commits Using Pre-Commit Hooks	66

6.3	Experimental Setup	67
6.3.1	Project Repository Selection	68
6.3.2	Project Dependency Analysis	68
6.3.3	Building a Database of Defect-Commits and Fix-Commits for Performances Evaluation	70
6.3.4	Process of Comparing New Commits	71
6.3.5	Evaluation Measures	72
6.4	Empirical Validation	73
6.4.1	Baseline Classifier Comparison	75
6.4.2	Performance of BIANCA	76
6.4.3	Analysis of the Quality of the Fixes Proposed by BIANCA . .	79
6.5	Threats to Validity	85
6.6	Chapter Summary	88
7	Combining Code Metrics with Clone Detection for Just-In-Time Fault Prevention and Resolution	89
7.1	Introduction	89
7.2	Approach	91
7.2.1	Clustering Projects	93
7.2.2	Building a Database of Code Blocks of Defect-Commits and Fix-Commits	93
7.2.3	Building a Metric-Based Model	95
7.2.4	Comparing Code Blocks	96
7.2.5	Classifying Incoming Commits	96
7.2.6	Proposing Fixes	96
7.3	Experimental Setup	97
7.3.1	Project Repository Selection	97
7.3.2	Project Dependency Analysis	98
7.3.3	Building a Database of Defect-Commits and Fix-Commits . .	98
7.3.4	Process of Comparing New Commits	98
7.4	Empirical Validation	99
7.4.1	Performance of CLEVER	99
7.4.2	Analysis of the Quality of the Fixes Proposed by CLEVER . .	100
7.5	Threats to Validity	103

7.6	Chapter Summary	104
8	Bug Reproduction Using Crash Traces and Directed Model Checking	105
8.1	Preliminaries	106
8.2	Approach	110
8.2.1	Collecting Crash Traces	111
8.2.2	Preprocessing	112
8.2.3	Building the Backward Static Slice	113
8.2.4	Directed Model Checking	118
8.2.5	Validation	120
8.2.6	Generating Test Cases for Bug Reproduction	121
8.3	Experimental Setup	123
8.3.1	Targeted Systems	123
8.3.2	Bug Selection and Crash Traces	126
8.4	Empirical Validation	126
8.4.1	Successfully Reproduced	128
8.4.2	Partially Reproduced	132
8.4.3	Not Reproduced	133
8.5	Threats to Validity	135
8.6	Chapter Summary	136
9	Towards a Classification of Bugs Based on the Location of the Corrections: An Empirical Study	138
9.1	Introduction	138
9.2	Experimental Setup	140
9.2.1	Context Selection	140
9.2.2	Dataset Analysis	142
9.3	Empirical Validation	150
9.3.1	Are T4 bug predictable at submission time?	150
9.3.2	What are the best predictors of type 4 bugs?	157
9.4	Threats to Validity	163
9.5	Chapter Summary	164

10 Discussion	165
10.1 Adoption of Fault Prevention Tools in Industry	165
10.2 The Right-Time For Just-In-Time Fault Prevention	167
10.2.1 Just-In-Time: From Toyota Plants to Software Quality	168
10.2.2 Types of Just-In-Time Quality Insurance	169
10.2.3 Conclusion	173
10.3 University-Industry Research Collaboration	174
10.3.1 Deep understanding of the project requirements	175
10.3.2 Understanding the benefits of the project to both parties	175
10.3.3 Focusing in the Beginning on Low-Hanging Fruits	176
10.3.4 Communicating effectively	176
10.3.5 Managing change	177
11 Conclusion and Future Work	178
11.1 Summary of the Findings	178
11.2 Future Work	180
11.2.1 Current Limitations	180
11.2.2 Other Possible Opportunities for Future Research	180
12 Appendices	182
12.1 Lists of the top-level open-source projects	182
12.1.1 Parsers	182
12.1.2 Databases	182
12.1.3 Web and Services	184
12.1.4 Cloud and Big data	186
12.1.5 Messaging and Logging	188
12.1.6 Graphics	188
12.1.7 Dependency Management and build systems	189
12.1.8 Networking	189
12.1.9 File systems and repository	189
12.1.10 Misc	190
Bibliography	214

List of Figures

1	Data structure of a commit.	12
2	Data structure of two commits.	12
3	Two branches pointing on one commit.	13
4	Two branches pointing on two commits.	13
5	Lifecycle of a report ?	14
6	Bumper Architecture	34
7	BUMPER Metamodel	35
8	Web Interface of BUMPER.	40
9	Overview of the PRECINCT approach.	45
10	PRECINCT output when replaying commit 710b6b4 of the Monit system used in the case study.	52
11	PRECINCT Branching.	54
12	Monit clone detection over revisions	56
13	JHotDraw clone detection over revisions	56
14	Dnsjava clone detection over revisions	57
15	Managing events happening on project tracking systems to extract defect-introducing commits and commits that provided the fixes . . .	63
16	Clustering by dependency	63
17	Classifying incoming commits and proposing fixes	64
18	Simplified Dependency Graph for <code>com.badlogicgames.gdx</code>	65
19	Dependency Graph	69
20	Precision, Recall and F_1 -measure variations according to α	71
21	okhttp commit #0ca4c82dd1032625831a5814ea2ddcf165029bdc	80
22	Druid commit #1091861bb15876131653191ae409a523aa8ec0c5	81
23	netty commit #085a61a310187052e32b4a0e7ae9700dbe926848	82
24	okhttp commit #a96c3a8007d8e1a166f7aec423c7add1ea0e3522	83

25	OrientDB commit #444db817ee9404b17c1208df51781ce9cb6a2666 . . .	85
26	Jsoup commit #6c4f16f233cdfd7aedef33374609e9aa4ede255c	86
27	Managing events happening on project tracking systems to extract defect-introducing commits and commits that provided the fixes . . .	92
28	Clustering by dependency	92
29	Classifying incoming commits and proposing fixes	92
30	Dependency Graph	94
31	Clusters	94
32	Process of Comparing New Commits	99
33	System with four steps S_0 to S_3 , which have five atomic properties p_1 to p_5	107
34	A toy program under testing	108
35	A toy program under model checking	109
36	A toy program under directed model checking	109
37	Overview of JCHARMING.	111
38	Java <code>InvalidActivityException</code> is thrown in the <code>Bar.Foo</code> loop if the control variable is greater than 2.	112
39	Hypothetical example of static program slicing	113
40	Hypothetical example of backward static program slicing	114
41	Hypothetical example representing $bslice_{[a \leftarrow d]}$ (left) vs. $\cup_{i=d}^a bslice_{[f_{i+1} \leftarrow f_i]}$ (right) for a crash trace $T = d, f, e$	115
42	Steps of Algorithm 2 where f_3 is corrupted.	117
43	High level architecture of the Junit test case generation	122
44	Simplified Unit Test template	124
45	Crash trace reported for bug ArgoUML #311	129
46	Crash trace reported for bug Mahout #486	131
47	Crash trace reported for bug JFreeChart #664	132
48	Class diagram showing the relationship between bugs and fixed . . .	139
49	Proposed Taxonomy of Bugs	139
50	Ambari heatmap	158

List of Tables

1	Resolved/Fixed Bug (R/F BR), Changesets (CS), and projects by dataset	38
2	Pretty-Printing Example	51
3	List of Target Systems in Terms of Files and Kilo Line of Code (KLOC) at current version and Language	53
4	Overview of PRECINCT's results in terms of precision, recall, F_1 - measure, execution time and output reduction.	57
5	Communities in terms of ID, Color code, Centroids, Betweenness and number of members	70
6	BIANCA results in terms of organization, project name, a short de- scription, number of class, number of commits, number of defect in- troducing commits, number of risky commit detected, precision (%), recall (%), F_1 -measure (%), the average similarity of first 3 and 5 pro- posed fixes with the actual fix and the average time difference between detected and original.	74
7	Workshop results	100
8	List of target systems in terms of Kilo line of code (KLoC), number of classes (NoC) and Bug # ID	125
9	Effectiveness of JCHARMING using directed model checking (DMC) in minutes, length of the generated JUnit tests (CE length) and model checking (MC) in minutes	127
10	Datasets	141
11	Contingency table and Pearson's chi-squared tests	142
12	Apache Ecosystem Complexity Metrics Comparison and Mann-whitney test results.	
	Σ :sum, \hat{x} :median, σ :standard deviation, %:percentage	144

13	Netbeans Ecosystem Complexity Metrics Comparison and Mann-whitney test results.	
	\sum :sum, \hat{x} :median, σ :standard deviation, %:percentage	145
14	Apache and Netbeans Ecosystems Complexity Metrics Comparison and Mann-whitney test results.	
	\sum :sum, \hat{x} :median, σ :standard deviation, %:percentage	146
15	Support Vector Machine, Linear Regression and Random Forest based classifiers performances while using 1 gram.	
	True positive, TN: True Negative, FN: False Negative, FP: False Positive	153
16	Support Vector Machine, Linear Regression and Random Forest based classifiers performances while using 2 grams.	
	True positive, TN: True Negative, FN: False Negative, FP: False Positive	154
17	Support Vector Machine, Linear Regression and Random Forest based classifiers performances while using 3 grams.	
	True positive, TN: True Negative, FN: False Negative, FP: False Positive	155
18	Random classifier.	
	True positive, TN: True Negative, FN: False Negative, FP: False Positive	156
19	Cohen's Kappa for each classifier	156

Chapter 1

Introduction

Software maintenance activities such as debugging and feature enhancement are known to be challenging and costly (Pressman 2005). Studies have shown that the cost of software maintenance can reach up to 70% of the overall cost of the software development life cycle (Health, Social and Research 2002). Much of this is attributable to several factors including the increase in software complexity, the lack of traceability between the various artifacts of the software development process, the lack of proper documentation, and the unavailability of the original developers of the systems.

More than three decades of research in different fields such mining software repository, default prevention, clone detection, program comprehension or software maintenance aimed to understand better the needs of and challenges faced by developers when maintaining a program. Researchers have many approaches and tools to improve the maintenance processes.

The large adoption of these tools by practitioners remains limited (Lewis et al. 2013; Foss and Murphy 2015; Layman, Williams, and Amant 2007; Ayewah et al. 2007; Johnson et al. 2013). Factors that prevent such adoption are still an open question. Nevertheless, based on developers interviews, several hypotheses have been formulated. For example, developers are known to have trust issues with statistical models based on process or code metrics. Another hypothesis for the limited adoption of software maintenance approaches is the lack of integration with developers' workflow. Finally, developers also express concerns regarding the numbers of warnings, the general heaviness of information provided by software maintenance tools and the

lack of clear corrective actions to fix a given warning.

In this thesis, we propose to address some of the issues mentioned above by focusing on developing techniques and tools that support software maintainers at commit-time. As part of the developer’s workflow, a commit marks the end of a given task or subtask as the developer is ready to version the source code. Commits are bite-sized units of work that are potentially ready to be shared with the rest of the organization (O’Sullivan and Bryan 2009).

We propose a set of approaches in which we intercept the commits and analyze them with the objective of preventing unwanted modifications to the system. By doing so, we do not only propose solutions that integrate well with the developer’s work flow, but also there is no need for software developers to use any other external tools.

1.1 Thesis Contributions

In this thesis, we make the following contributions:

- We create a metamodel that allows researchers to manipulate millions of bug reports and fixes (Chapter 4).

In this work, we introduce BUMPER (BUg Metarepository for dEvelopers and Researchers), a web-based infrastructure that can be used by software developers and researchers to access data from diverse repositories using natural language queries transparently, regardless of where the data was created and hosted (Nayrolles and Hamou-Lhadj 2016). The idea behind BUMPER is that it can connect to any bug tracking and version control systems and download the data into a single database. We created a common schema that represents data, stored in various bug tracking and version control systems. BUMPER uses a web-based interface to allow users to search the aggregated database by expressing queries through a single point of access. This way, users can focus on the analysis itself and not on the way the data is represented or located. BUMPER supports many features including: (1) the ability to use multiple bug tracking and control version systems, (2) the ability to search very efficiently large data repositories using both natural language and a specialized query language, (3) the mapping between the bug reports and the fixes, and (4) the ability to export the search results in JSON, CSV and XML formats.

Importantly, BUMPER differs from other approaches such as Boa (Dyer et al. 2013) because (a) it updates itself every day with the new closed reports, (b) it proposes a clear and concise JSON API that anyone can use to support their approaches or tools.

- We study online and incremental clone detection at commit-time that build upon existing offline techniques (Chapter 5).

Code clones appear when developers reuse code with little to no modification to the original code. Studies have shown that clones can account for about 7% to 50% of the code in a given software system (Baker 1995; Ducasse, Rieger, and Demeyer 1999). Nevertheless, clones are considered a bad practice in software development since they can introduce new bugs in code (Juergens et al. 2009). If a bug is discovered in one segment of the code that has been copied and pasted several times, then the developers will have to remember the places where this segment has been reused to fix the bug in each place.

In this research, we present PRECINCT (PREventing Clones INsertion at Commit-Time) that focuses on preventing the insertion of clones at commit time, i.e., before they reach the central code repository. PRECINCT is an online clone detection technique that relies on the use of pre-commit hooks capabilities of modern source code version control systems.

A pre-commit hook is a process that one can implement to receive the latest modification to the source code done by a given developer just before the code reaches the central repository. PRECINCT intercepts this modification and analyses its content to see whether a suspicious clone has been introduced or not. A flag is raised if a code fragment is suspected to be a clone of an existing code segment. In fact, PRECINCT, itself, can be seen as a pre-commit hook that detects clones that might have been inserted in the latest changes with regard to the rest of the source code.

- We use incremental clone detection to prevent bug insertion and provide possible fixes (Chapter 6).

Similar to clone detection, we propose an approach for preventing the introduction of bugs at commit-time. Many tools exist to prevent a developer to ship *bad* code (Hovemeyer 2007).

Our approach, called BIANCA (Bug Insertion ANTicipation by Clone Analysis at commit-time), is different than the approaches presented in the literature because it mines and analyses the change patterns in commits and matches them against past commits known to have introduced a defect in the code (or that have just been replaced by better implementation).

- We address scalability and efficiency issues of clone-based bug detection and resolution approaches by combining them with metric based approaches (Chapter 7).

Clone-based bug detection approaches suffer from scalability issues as Type 2, and Type 3 clone-comparison are expensive computation wise. In industrial settings, where several repositories receive hundreds of commits per day they are not applicable.

We created a two-step classifier that leverage the performances of metric based detection and the expressiveness of clone-base detection and resolution called CLEVER (Nayrolles and Hamou-lhadj 2018).

This work has been conducted in partnership with Ubisoft, one of the largest video games company in the world.

- We present an approach for crash reproductions in a controlled environment that can be used if the aforementioned approaches aiming to avoid defect introduction failed (Chapter 8).

When preventing measures have failed, and bugs have to be fixed; the first step is to reproduce what happened on sites. Crash reproduction is an expensive task because the data provided by end users is often scarce (N. Chen 2013a). It is, therefore, important to invest in techniques and tools for automatic bug reproduction to ease the maintenance process and accelerate the rate of bug fixes and patches. Existing techniques can be divided into two categories: (a) On-field record and in-house replay (Roehm, Nosovic, and Bruegge 2015), and (b) In-house crash explanation (Zuddas et al. 2014).

We propose an approach, called JCHARMING (Java CrasH Automatic Reproduction by directed Model checkING) that uses a combination of crash traces and model checking to reproduce bugs that caused field failures automatically (Mathieu Nayrolles, Hamou-Lhadj, et al. 2015; Nayrolles et al. 2016). Unlike existing techniques, JCHARMING does not require instrumentation of the code. It does not need

access to the content of the heap either. Instead, JCHARMING uses a list of functions output when an uncaught exception in Java occurs (i.e., the crash trace) to guide a model checking engine to uncover the statements that caused the crash. Such outputs are often found in bug reports.

- We propose a taxonomy of bugs based on the location of their fixes based on a two software ecosystems (Chapter 9).

In recent years, there has been an increase in attention in techniques and tools that mine large bug repositories to help software developers understand the causes of bugs and speed up the fixing process. These techniques, however, treat all bugs in the same way. Bugs that are fixed by changing a single location in the code are examined the same way as those that require complex changes. After examining more than 100 thousand bug reports of 380 projects, we found that bugs can be classified into four types based on the location of their fixes. Type 1 bugs are the ones that fixed by modifying a single location in the code, while Type 2 refers to bugs that are fixed in more than one location. Type 3 refers to multiple bugs that are fixed in the exact same location. Type 4 is an extension of Type 3, where multiple bugs are resolved by modifying the same set of locations. This classification can help companies put the resources where they are needed the most. It also provides useful insight into the quality of the code. Knowing, for example, that a system contains a large number of bugs of Type 4 suggests high maintenance efforts. This classification can also be used for other tasks such as predicting the type of incoming bugs for an improved bug handling process. For example, if a bug is found to be of Type 4, then it should be directed to experienced developers.

1.2 Thesis Organization

This thesis organization is as follows; in Chapter 2, we provide background information about the version control systems and project tracking systems. In Chapter 3, we present the works related to ours. The works related to clone detection, fault-detection, fault-fixing and crash reproduction is extensive as researchers have been focusing on these areas for more than 20 years. We focus, however, on the works that are the most related to ours. Chapters 4, 5, 6, 7, 8 and, 9 are dedicated to the main

contributions of this thesis we mentioned in the previous section. Finally, Chapter 10 and 11 present a discussion that reflects on the entirety of our works and a conclusion accompanied by avenues for future works, respectively.

1.3 Related Publications

Earlier versions of the work done in this thesis have been published in the following papers:

1. Abdelwahab Hamou-Lhadj, **Mathieu Nayrolles**: A Project on Software Defect Prevention at Commit-Time: A Success Story of University-Industry Research Collaboration. Accepted to SER&IP 2018 (Co-located with ICSE'18).
2. **Mathieu Nayrolles**, Abdelwahab Hamou-Lhadj: CLEVER: Combining Code Metrics with Clone Detection for Just-In-Time Fault Prevention and Resolution in Large Industrial Projects. Accepted to MSR 2018 (Co-located with ICSE'18).
3. **Mathieu Nayrolles**, Abdelwahab Hamou-Lhadj: Towards a Classification of Bugs to Facilitate Software Maintainability Tasks. Accepted to SQUADE 2018 (Co-located with ICSE'18).
4. **Mathieu Nayrolles**, Abdelwahab Hamou-Lhadj, Sofine Tahar, Alf Larsson: A bug reproduction approach based on directed model checking and crash traces. Journal of Software: Evolution and Process 29(3) (2017).
5. **Mathieu Nayrolles**, Abdelwahab Hamou-Lhadj: BUMPER: A Tool for Coping with Natural Language Searches of Millions of Bugs and Fixes. SANER 2016: 649-652.
6. **Mathieu Nayrolles**, Abdelwahab Hamou-Lhadj, Sofine Tahar, Alf Larsson: JCHARMING: A bug reproduction approach using crash traces and directed model checking. SANER 2015: 101-110. **Best Paper Award**.

The following papers were published in parallel to the aforementioned publications. While they are not directly related to this thesis, at the same time, they are not completely irrelevant, as their topics include crash report handling and quality oriented refactoring of service based applications.

7. Abdou Maiga, Abdelwahab Hamou-Lhadj, **Mathieu Nayrolles**, Korosh Koochekian Sabor, Alf Larsson: An empirical study on the handling of crash reports in a large software company: An experience report. ICSME 2015: 342-351.
8. **Mathieu Nayrolles**, Eric Beaudry, Naouel Moha, Petko Valtchev, Abdelwahab Hamou-Lhadj: Towards Quality-Driven SOA Systems Refactoring Through Planning. MCETECH 2015: 269-284.

In addition, we seized the opportunity to disseminate the best practices discovered from our extensive investigation of software ecosystems in several books aimed at practitioners. Appendices of this thesis list the open-source systems that have been studied for our works.

9. **Mathieu Nayrolles**, Rajesh Gunasundaram, Sridhar Rao (2017). Expert Angular. (pp. 454). Packt Publishing.
10. **Mathieu Nayrolles** (2015). Magento Site Performance Optimization. (pp. 92). Packt Publishing.
11. **Mathieu Nayrolles** (2015). Xamarin Studio for Android Programming: A C# Cookbook. (pp. 298). Packt Publishing.
12. **Mathieu Nayrolles** (2014). Mastering Apache Solr. Inkstall Publishing. (pp. 152). Inkstall Publishing.
13. **Mathieu Nayrolles** (2013). Instant Magento Performance Optimization How-to. (pp. 56). Packt Publishing.

Finally, the work presented in this thesis also attracted media-coverage for its impact at Ubisoft, one of the world largest video game publisher. A google search for “commit+assistant+ubisoft” yields more than 114,000 results at the time of writing. Here is a curated list of the most interesting press articles.

14. Sinclair, B. (2018). Ubisoft’s “Minority Report of programming” - GamesIndustry. <https://www.gamesindustry.biz/articles/2018-02-22-ubisofts-minority-report-of-programming>.

15. Maxime Johnson. (2018). Jeux videos : reunir les chercheurs et les createurs - Techno - L'actualite. <http://lactualite.com/techno/2018/02/23/jeu-video-reunir-les-chercheurs-et-les-createurs/>
16. Condliffe, J. (2018). AI can help spot coding mistakes before they happen. - MIT Technology Review. <https://www.technologyreview.com/the-download/610416/ai-can-help-spot-coding-mistakes-before-they-happen/>
17. Matt Kamen. (2018). Ubisoft's AI in Far Cry 5 and Watch Dogs could change gaming - WIRED UK.<http://www.wired.co.uk/article/ubisoft-commit-assist-ai>
18. Kenneth Gibson. (2018). STEM SIGHTS: The Concordian who uses AI to fix software bugs - Concordia News. <http://www.concordia.ca/cunews/main/stories/2018/04/10/stem-sights-concordian-who-makes-bug-free-software.html>
19. Ryan Remiorz. (2018). Concordia develops tool with Ubisoft to detect glitches in gaming software - Financial Post. <http://business.financialpost.com/pmn/business-pmn/concordia-develops-tool-with-ubisoft-to-detect-glitches-in-gaming-software>
20. The Canadian Press. (2018). Concordia partners with Ubisoft to detect glitches in gaming software - The Globe and Mail. <https://www.theglobeandmail.com/business/technology/article-concordia-partners-with-ubisoft-to-detect-glitches-in-gaming-software/>
21. Cyrille Baron. (2018). Commit Assistant, l'IA qui aide les dveloppeurs de jeux - iQ France. <https://iq.intel.fr/commit-assistant-lia-qui-aide-les-developpeurs-de-jeux/?sf184907379=1>

In these publications, the work presented in this thesis is referred to as *commit-assistant* which is the internal implementation of CLEVER (Chapter 7).

Chapter 2

Background

2.1 Definitions

In this thesis, we use the following definitions that are based on (Avizienis et al. 2004; Pratt 2001; Burnstein 2006; Radatz, Geraci, and Katki 1990; Whittaker, Arbon, and Carollo 2012).

- **Software bug:** A software bug is an error, flaw, failure, defect or fault in a computer program or system that causes it to violate at least one of its functional or non-functional requirement.
- **Error:** An error is a mistake, misconception, or misunderstanding on the part of a software developer.
- **Fault/defect:** A fault (defect) is defined as an abnormal condition or defect at the component, equipment, or subsystem level which may lead to a failure. A fault (defect) is not final (the system still works) and does not prevent a given feature to be accomplished. A fault (defect) is a deviation (anomaly) of the healthy system that can be caused by an error or external factors (hardware, third parties, etc.).
- **Failure:** The inability of a software system or component to perform its required functions within specified requirements.
- **Crash:** The software system encountered a fault (defect) that triggered a fatal failure from which the system could not recover from/overcome. As a result, the system stops.

- Bug report: A bug report describes a behaviour observed in the field and considered abnormal by the reporter. Bug reports are submitted manually to bug report systems (bugzilla/jira). There is no mandatory format to report a bug. Nevertheless, a bug report should have the version of the software system, OS, and platform, steps to reproduce the bug, screen shots, stack trace and anything that could help a developer assess the internal state of the software system.
- Crash report: A crash report is issued as the last thing that a software system does before crashing. Crash reports are usually reported automatically (crash reporting systems are implemented as part of the software). A crash report contains data (that can be proprietary) to help developers understand the causes of the crash (e.g., memory dump, ...).

In the remaining of this section, we introduce the two types of software repositories: version control and project tracking system.

2.2 Version control systems

Version control consists of maintaining the versions of various artifacts such as source code files (Zeller 1997). This activity is a complex task and cannot be performed manually in real world projects. To this end, there exist several tools that have been created to help practitioners manage the version of their software artifacts. Each evolution of a software system is considered as a version (also called revision) and each version is linked to the one before through modifications of software artifacts. These modifications consist of updating, adding or deleting software artifacts. They can be referred as diff, patch or commit¹. A diff, patch or commit has the following characteristics:

- Number of files: The number of software files that have been modified, added or deleted
- Number of hunks: The number of consecutive code blocks of modified, added or deleted lines in textual files. Hunks are used to determine, in each file, how many different places the developer has modified
- Number of churns: The number of modified lines. However, the churn value for a line change should be at least two as the line has to be

¹These names are not to be used interchangeably as differences exists.

Modern version control systems also support branching. A branch is a derivation in the evolution that contains a duplication of the source code so that both versions can be modified in parallel. Branches can be reconciled with a merge operation that merges modifications of two or more branches. This operation is completely automated with the exception of merging conflicts that arise when both branches contain modifications of the same line. Such conflicts cannot be reconciled automatically and have to be dealt with by the developers. This allows for greater agility among developers as changes in one branch do not affect the work of the developers that are on other branches.

Branching has been used for more than testing hazardous refactoring or testing framework upgrades. Task branching is an agile branching strategy where a new branch is created for each task (Martin Fowler 2009). It is common to see a branch named `123_implement_X` where 123 is the #id of task X given by the project tracking system. Project tracking systems are presented in Section 2.3.1.

In modern versioning systems, when maintainers make modifications to the source code, they have to commit their changes for the modifications to be effective. The commit operation versions the modifications applied to one or many files.

Figure 1 presents the data structure used to store a commit. Each commit is represented as a tree. The root leaf (green) contains the commit, tree and parent hashes as same as the author and the description associated with the commit. The second leaf (blue) contains the leaf hash and the hashes of the files of the project.

In this example, we can see that author “Mathieu” has created the file `file1.java` with the message “project init”. Figure 2 represents an external modification. In this second example, `file1.java` is modified while `file2.java` is created. The second commit 98ca9 have 34ac2 as a parent.

Branches point to a commit. In a task-branching environment, a branch is created via a checkout operation for each task. Tasks can be to fix the root cause of a crash or bug report or features to implement. In figure 3, the master branch and the `1_fix_overflow` point on commit 98ca9.

Both branches can evolve separately and be merged when the task branch is ready. In Figure 4, the master branch points on a13ab2 while the `1_fix_overflow` points on ahj23k.

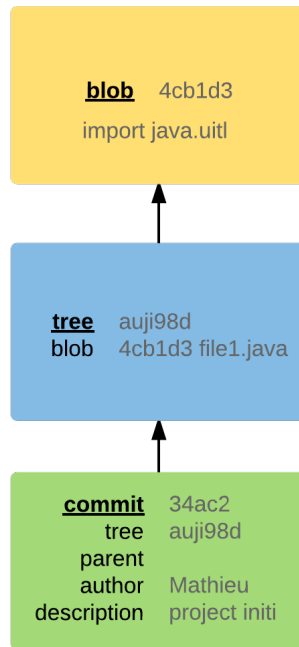


Figure 1: Data structure of a commit.

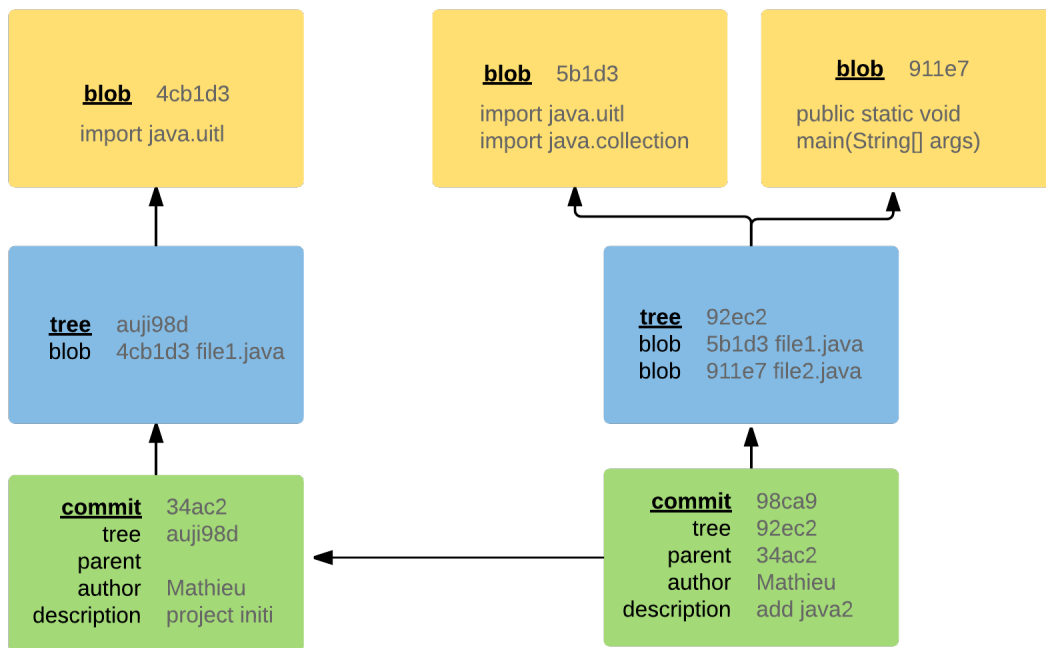


Figure 2: Data structure of two commits.

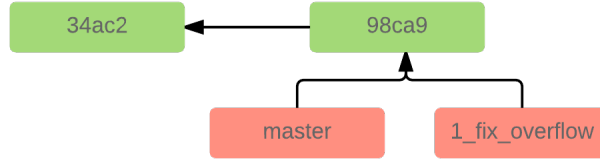


Figure 3: Two branches pointing on one commit.

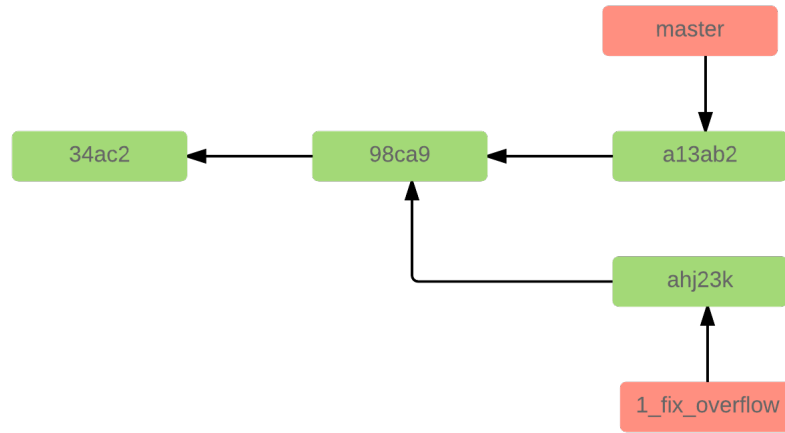


Figure 4: Two branches pointing on two commits.

2.3 Providers

In this proposal, we mainly refer to three version control systems: Svn, Git and, to a lesser extent, Mercurial. SVN is distributed by the Apache Foundation and is a centralized concurrent version system that can handle conflicts in the different versions of different developers. SVN is widely used in industry. At the opposite, Git is a distributed revision control system — originally developed by Linus Torvald — where revisions can be kept locally for a while and then shared with the rest of the team. Finally, Mercurial is also a distributed revision system, but shares many concepts with SVN. It will be easier for people who are used to SVN to switch to a distributed revision system if they use Mercurial.

2.3.1 Project Tracking Systems

Project tracking systems allow end users to create bug reports (BRs) to report unexpected system behaviour, managers can create tasks to drive the evolution forward

and crash report (CRs) can be automatically created. These systems are also used by development teams to keep track of the modifications induced by bugs, crash reports, and keep track of the fixes.

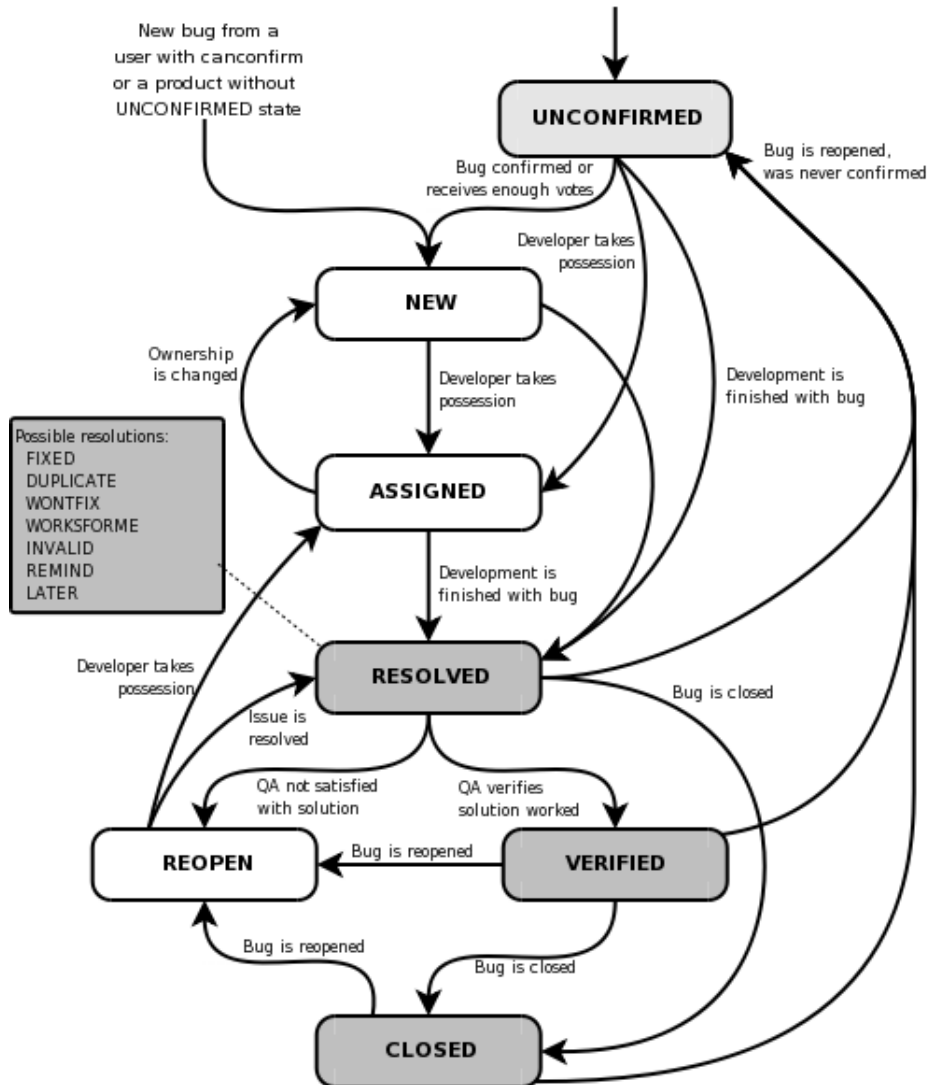


Figure 5: Lifecycle of a report ?

Figure 5 presents the life cycle of a report. When an end-user submits a report, it is set to the UNCONFIRMED state until it receives enough votes or that a user with the proper permissions modifies its status to NEW. The report is then assigned to a developer to be fixed. When the report is in the ASSIGNED state, the assigned developer(s) starts working on the report. A fixed report moves to the RESOLVED state. Developers have five different possibilities to resolve a report: FIXED,

DUPLICATE, WONTFIX, WORKSFORME and INVALID (Koponen 2006).

- RESOLVED/FIXED: A modification to the source code has been pushed, i.e., a changeset (also called a patch) has been committed to the source code management system and fixes the root problem described in the report.
- RESOLVED/DUPLICATE: A previously submitted report is being processed. The report is marked as a duplicate of the original report.
- RESOLVED/WONTFIX: This is applied in the case where developers decide that a given report will not be fixed.
- RESOLVED/WORKSFORME: If the root problem described in the report cannot be reproduced on the reported OS/hardware.
- RESOLVED/INVALID: If the report is not related to the software itself.

Finally, the report is CLOSED after it is resolved. A report can be reopened (sent to the REOPENED state) and then assigned again if the initial fix was not adequate (the fix did not resolve the problem). The elapsed time between the report marked as the new one and the resolved status is known as the *fixing time*, usually in days. In case of task branching, the branch associated with the report is marked as ready to be merged. Then, the person in charge (quality assurance team, manager, ect...) will be able to merge the branch with the mainline. If the report is reopened: the days between the time the report is reopened and the time it is marked again as RESOLVED/FIXED are cumulated. Reports can be reopened many times.

Tasks follow a similar life cycle with the exception of the UNCONFIRMED and RESOLVED states. Tasks are created by management and do not need to be confirmed to be OPEN and ASSIGNED to developers. When a task is complete, it will not go to the RESOLVED state, but to the IMPLEMENTED state. Bug and crash reports are considered as problems to eradicate in the program. Tasks are considered as new features or amelioration to include in the program.

Reports and tasks can have a severity associated with them (Bettenburg et al. 2008). The severity indicates the degree of impact on the software system. The possible severities are:

- blocker: blocks development and/or testing work.
- critical: crashes, loss of data, severe memory leak.
- major: major loss of function.

- normal: regular report, some loss of functionality under specific circumstances.
- minor: minor loss of function, or other problem where easy workaround is present.
- trivial: cosmetic problems like misspelled words or misaligned text.

The relationship between a report or a task and the actual modification can be hard to establish, and it has been a subject of various research studies (e.g., (Antoniol et al. 2002; Bachmann et al. 2010; Wu et al. 2011)). The reason is that they are on two different systems: the version control system and the project tracking system. While it is considered a good practice to link each report with the versioning system by indicating the report #id on the modification message, more than half of the reports are not linked to a modification (Wu et al. 2011).

2.4 Providers

We have collected data from four different project tracking systems: Bugzilla, Jira, Github and Sourceforge. Bugzilla belongs to the Mozilla foundation and has first been released in 1998. Jira, provided by Atlassian, has been released 14 years ago, in 2002. Bugzilla is 100% open source and it is difficult to estimate how many projects use it. However, we can envision that it owns a great share of the market as major organizations such as Mozilla, Eclipse and the Apache Software Foundation use it. Jira, on the other hand, is a commercial software tool — with a freemium business model — and Atlassian claims that they have 25,000 customers over the world.

Github and Sourceforge are different from Bugzilla and Jira in a sense that they were created as source code revision systems and evolved, later on, to add project tracking capabilities to their software tools. This common particularity has the advantage to ease the link between bug reports and the source code.

Chapter 3

Related work

3.1 Clone Detection

Clone detection is an important and difficult task. Throughout the years, researchers and practitioners have developed a considerable number of methods and tools to efficiently detect source code clones. In this section, we first describe the classical clone detection approaches and then present works that focus on local and remote detection.

3.1.1 Traditional Clone Detection Techniques

Tree-matching and metric-based methods are two sub-categories of syntactic analysis for clone detection. Syntactic analyses consist of building abstract syntax trees (AST) and analyze them with a set of dedicated metrics or searching for identical sub-trees. Many existing AST-based approaches rely on sub-tree comparison to detect clone, including the work of Baxter et al. (Baxter et al. 1998), Wahleret et al. (Wahler et al. 2004), and the work of Jian et al. with Deckard (Jiang et al. 2007). An AST-based approach compares metrics computed on the AST, rather than the code itself, to identify clones (Paternaude et al. 1999; Balazinska et al. 1999).

Text-based techniques use the code and compare sequences of code blocks to each other to identify potential clones. Johnson was perhaps the first one to use fingerprints to detect clones (Johnson 1993; Johnson 1994). Blocks of code are hashed, producing fingerprints that can be compared. If two blocks share the same fingerprint, they are considered as clones. Manber et al. (Manber 1994) and Ducasse et al. (Ducasse,

Rieger, and Demeyer 1999) refined the fingerprint technique by using leading keywords and dot-plots.

Another approach for detecting clones is to use static analysis and to leverage the semantics of a program to improve the detection. These techniques rely on program dependency graphs, where nodes are statements and edges are dependencies. Then, the problem of finding clones is reduced to the problem of finding identical sub-groups in the program dependency graph. Examples of existing techniques that fall into this category are the ones presented by Krinke et al. (Krinke 2001) and Gabel et al. (Gabel, Jiang, and Su 2008).

Many clone detection tools resort to lexical approaches for clone detection. Here, the code is transformed into a series of tokens. If sub-series repeat themselves, it means that a potential clone is in the code. Some popular tools that use this technique include Dup (Baker 1995), CCFinder (Kamiya, Kusumoto, and Inoue 2002), and CP-Miner (Li et al. 2006).

In 2010, Hummel et al. proposed an approach that is both incremental and scalable using index-based clone detection (Hummel et al. 2010). Incremental clone detection is a technique where only the changes from one version to another are analysed. Thus, the required computational time is greatly reduced. Using more than 100 machines in a cluster, they managed to drop the computation time of Type 1 and 2 to less than a second while comparing a new version. The time required to find all the clones on a 73 MLOC system was 36 minute. We reach similar performances, for one revision, using a single machine. While being extremely fast and reliable, Hummel et al.’s approach required an industrial cluster to achieve such performance. In our opinion, it is unlikely that standard practitioners have access to such computational power. Moreover, the authors’ approach only targets Type 1 and 2 clones. Higo et al. proposed an incremental clone detection approach based on program dependency graphs (PDG) (Higo et al. 2011). Using PDG is arguably more complex than text comparison and allows the detection of clone structures that are scattered in the program. They were able to analyze 5,903 revisions in 15 hours in Apache Ant.

3.1.2 Remote Detection of Clones

Yuki et al. conducted one of the few studies on the application of clone management to industrial systems (Yamanaka et al. 2012). They implemented a tool named

Clone Notifier at NEC with the help of experienced practitioners. They specifically focus on clone insertion notification, very much like PRECINCT. Unlike PRECINCT, their approach uses a remote approach in which the changes are committed (i.e., they reach the central repository, and anyone can pull them into his or her machines) and a central server analyses the changes. If the committed changes contain newly inserted clones, then an email notification is sent.

Zhang et al. proposed CCEvents (Code Cloning Events) (Zhang et al. 2013). Their approach monitors code repository continuously and allows stakeholders to use a domain specific language called CCEML to specify which email notifications they wish to receive.

In addition, many commercial tools now include clone detection as part of continuous integration. Codeclimate [^]codeclimate, Codacy [^]codacy, Scrutinizer [^]scrutinizer and Coveralls [^]coveralls are some examples. These tools will perform various tasks such as executing unit test suites, computing quality metrics performing clone detection and, provide a report by email.

We argue that remotely detecting clones is not practical because clones can be synchronized by other team members, which may lead to challenging merges when the clones are removed. In addition, the authors did not report performance measurements and the longer it takes for the notification to be sent to the developer, the harder it can be to reconstruct the mind-map required for clone removal.

3.1.3 Local Detection of Clones

Gode and Koschke (Gde and Koschke 2009) proposed an incremental clone detector that relies on the results of analysis from past versions of a system to only analyze the new changes. Their clone detector takes the form of an IDE plugin that alerts developers as soon as a clone is inserted into the program.

Zibran and Roy (Zibran and Roy 2011; Zibran and Roy 2012) proposed another IDE-based clone management system to detect and refactor near-miss clones for Eclipse. Their approach uses a k-difference hybrid suffix tree algorithm. It can detect clones in real-time and propose a semi-automated refactoring process.

Robert et al. (Tairas, Gray, and Baxter 2006) proposed another IDE plugin for Eclipse called CloneDR based on ASTs that introduced novel visualization for clone detection such as scatter-plots.

IDE-based methods tend to issue many warnings to developers that may interrupt their work, hence hindering their productivity (Ko et al. 2006). In addition, Latoza et al. (Latoza, Venolia, and DeLine 2006) found that there exist six different reasons that trigger the use of clones (e.g., copy and paste of code examples, reimplementations of the same functionality in different languages, etc.). Developers are aware that they are creating clones in five out of six situations. In such cases, warnings provided by IDE-based local detection techniques can be quite disturbing.

Nguyen et al. (Nguyen et al. 2009) proposed an advanced clone-aware source code management system called *Clever*. Their approach uses abstract syntax trees to detect, update, and manage clones. While efficient, their approach does not prevent the introduction of clones, and it is not incremental. Developers have to run a project-wide detection for each version of the program. The same teams (Nguyen et al. 2012) conducted follow-up study by making *Clever* incremental. Their new tool, *JSync*, is an incremental clone detector that will only perform the detection of clones on the new changes.

Niko et al. (Niko, Mircea, and Romain 2012) proposed techniques revolving around hashing to obtain a quick answer while detecting Type 1, Type 2, and Type 3 clones in Squeaksource. While their approach works on a single system (i.e., detecting clones on one version of one system), they found that more than 14% of all clones are copied from project to project, stressing the need for fast and scalable approaches for clone detection to detect clone across a large number of projects. On the performance side, Niko et al. were able to perform clone detection on 74,026 classes in 14:45 hours (4,747 class per hour) with an eight core Xeon at 2.3 GHz with 16 GB of RAM. While these results are promising, especially because the approach detects clones across projects and versions, the computing power required is still considerable.

Similarly, Saini et al. (Saini et al. 2016) and Sajnani et al. (Sajnani et al. 2016) proposed an approach, called *SourcererCC*. *SourcererCC* targets fast clone detection on developers' workstation (12 GB RAM). *SourcererCC* is a token-based clone detector that uses an optimized inverted-index. It was tested on 25K projects cumulating 250 MLOC. The technique achieves a precision of 86% and a recall of 86%-100% for clones of Type 1, 2 and 3.

Toomey et al. (Toomey 2012) also proposed an efficient token based approach for detecting clones called *ctcompare*. Their tokenization is, however, different than most

approaches as they used lexical analysis to produce sequences of tokens that can be transformed into token tuples. `ctcompare` is accurate, scalable and fast but does not detect Type 3 clones.

3.2 Reports and source code relationships

Mining bug repositories is perhaps one of the most active research fields today. The reason is that the analysis of bug reports (BRs) provides useful insight that can help with many maintenance activities such as bug fixing (Wei, Zimmermann, and Zeller 2007; Saha, Khurshid, and Perry 2014) bug reproduction (N. Chen 2013a; Artzi, Kim, and Ernst 2008; Jin and Orso 2012), fault analysis (Nessa et al. 2008), etc. This increase of attention can be further justified by the emergence of many open source bug tracking systems, allowing software teams to make their bug reports available online to researchers.

These studies, however, treat all bugs as the same. For example, a bug that requires only one fix is analyzed the same way as a bug that necessitates multiple fixes. Similarly, if multiple bugs are fixed by modifying the exact same locations in the code, then we should investigate how these bugs are related in order to predict them in the future.

Researchers have been studying the relationships between the bug and source code repositories since more than two decades. To the best of our knowledge, the first ones who conducted this type of study on a significant scale were Perry and Stieg (Perry, Dewayne E. and Stieg. 1993). In these two decades, many aspects of these relationships have been studied in length. For example, researchers were interested in improving the bug reports themselves by proposing guidelines (Bettenburg et al. 2008), and by further simplifying existing bug reporting models (Herraiz et al. 2008).

Another field of study consists of assigning these bug reports, automatically if possible, to the right developers during triaging (Anvik, Hiew, and Murphy 2006; Jeong, Kim, and Zimmermann 2009; Tamrawi et al. 2011; Bortis and Hoek 2013). Another set of approaches focus on how long it takes to fix a bug (Bhattacharya and Neamtiu 2011; Zhang, Gong, and Versteeg 2013; Saha, Khurshid, and Perry 2014) and where it should be fixed (Zeller 2013; Zhou, Zhang, and Lo 2012). With the rapidly increasing number of bugs, the community was also interested in prioritizing

bug reports (Kim et al. 2011), and in predicting the severity of a bug (Lamkanfi et al. 2010). Finally, researchers proposed approaches to predict which bug will get reopened [Zimmermann et al. (2012); Lo2013], which bug report is a duplicate of another one (Bettenburg, Premraj, and Zimmermann 2008; Tian, Sun, and Lo 2012; Jalbert and Weimer 2008) and which locations are likely to yield new bugs [Kim, Zimmermann, Pan, et al. (2006a); Kim2007a].

3.3 Fault Prediction

The majority of previous file/module-level prediction work used code or process metrics. Approaches using code metrics only use information from the code itself and do not use any historical data. Chidamber and Kemerer published the well-known CK metrics suite (Chidamber and Kemerer 1994) for object oriented designs and inspired Moha et al. to publish similar metrics for service-oriented programs (Moha et al. 2012). Another famous metric suite for assessing the quality of a given software design is Briand’s coupling metrics (Briand, Daly, and Wust 1999).

The CK and Briand’s metrics suites have been used, for example, by Basili et al. (Basili, Briand, and Melo 1996), El Emam et al. (El Emam, Melo, and Machado 2001), Subramanyam et al. (Subramanyam and Krishnan 2003) and Gyimothy et al. (Gyimothy, Ferenc, and Siket 2005) for object-oriented designs. Service oriented designs have been far less studied than object oriented design as they are relatively new, but, Nayrolles et al. (Nayrolles et al., n.d.; Nayrolles, Moha, and Valtchev 2013), Demange et al. (Demange, Moha, and Tremblay 2013) and Palma et al. (Palma 2013) used Moha et al. metric suites to detect software defects. All these approaches, proved software metrics to be useful at detecting software fault for object oriented and service oriented designs, respectively. More recently, Nagappan et al. (Nagappan and Ball 2005; Nagappan, Ball, and Zeller 2006) and Zimmerman et al. (Zimmermann, Premraj, and Zeller 2007; Zimmermann and Nagappan 2008) further refined metrics-based detection by using statical analysis and call-graph analysis.

Other approaches use historical development data, often referred to as process metrics. Naggapan and Ball (N. Nagappan and Ball 2005) studied the feasibility of using relative churn metrics to prediction buggy modules in the Windows Server 2003. Other work by Hassan et al. and Ostrand et al. used past changes and defects

to predict buggy locations (e.g., (Hassan and Holt 2005), (Ostrand, Weyuker, and Bell 2005)). Hassan and Holt proposed an approach that highlights the top ten most susceptible locations to have a bug using heuristics based on file-level metrics (Hassan and Holt 2005). They find that locations that have been recently modified and fixed locations are the most defect-prone. Similarly, Ostrand et al. (Ostrand, Weyuker, and Bell 2005) predict future crash location by combining the data from changed and past defect locations. They validate their approach on industrial systems at AT&T. They showed that data from prior changes and defects can effectively predict defect-prone locations for open-source and industrial systems. Kim et al. (Kim, Zimmermann, Whitehead Jr., et al. 2007a) proposed the bug cache approach, which is an improved technique over Hassan and Holt’s approach (Hassan and Holt 2005). Rahman and Devanbu found that, in general, process-based metrics perform as good as or better than code-based metrics (Rahman and Devanbu 2013).

Other work focused on the prediction of risky changes. Kim et al. proposed the change classification problem, which predicts whether a change is buggy or clean (Sunghun Kim, Whitehead, and Yi Zhang 2008). Hassan (Hassan 2009) used the entropy of changes to predict risky changes. They find that the more complex a change is, the more likely it is to introduce a defect. Kamei et al. performed a large-scale empirical study on change classification (Kamei et al. 2013). The studies above find that size of a change and the history of the files being changed (i.e., how buggy they were in the past) are the best indicators of risky changes.

3.4 Automatic Patch Generation

Pan et al. (Pan, Kim, and Whitehead 2008) identified 27 bug fixing patterns that can be applied to fix software bugs in Java programs. They showed that between 45.7 - 63.6% of the bugs could be fixed with their patterns. Later, Kim et al. (Kim et al. 2013) generated patches from human-written patches and showed that their tool, PAR, successfully generated patches for 27 of 119 bugs. Tao et al. (Tao et al. 2014) also showed that automatically generated patches can assist developers in debugging tasks. Other work also focused on determining how to best generate acceptable and high quality patches, e.g. (Dallmeier, Zeller, and Meyer 2009; Le Goues et al. 2012), and determine what bugs are best fit for automatic patch generation (Le, Le, and Lo

2015).

Our work differs from the work on automated patch generation in that we do not generate patches; rather we use clone detection to determine the similarity of a change to a previous risky change and suggest to the developer the fixes of the prior risky changes.

3.5 Crash Reproduction

In this section, we put the emphasis on how crash traces are used in crash reproduction tasks. Existing studies can be divided into two distinct categories: (A) on-field record and in-house replay techniques (Steven et al. 2000; Narayanasamy, Pokam, and Calder 2005; Artzi, Kim, and Ernst 2008; Roehm, Nosovic, and Bruegge 2015), and (B) on-house crash understanding (Jin and Orso 2012; Jin and Orso 2013; Zuddas et al. 2014; N. Chen 2013b; Mathieu Nayrolles, Hamou-Lhadj, et al. 2015).

These two categories yield varying results depending on the selected approach and are mainly differentiated by the need for instrumentation. The first category of techniques oversees – by means of instrumentation – the execution of the target system on the field in order to reproduce the crashes in-house, whereas tools and approaches belonging to the second category only use data produced by the crash such as the crash stack or the core dump at crash time. In the first category, tools record different types of data such as the invoked methods (Narayanasamy, Pokam, and Calder 2005), try-catch exceptions (Rler et al. 2013), or objects (Jaygarl et al. 2010). In the second category, existing tools and approaches are aimed towards understanding the causes of a crash, using data produced by the crash itself, such as a crash stack (N. Chen 2013b), previous – and controlled – execution (Zuddas et al. 2014), etc.

Tools and approaches that rely on instrumentation face common limitations such as the need to instrument the source code in order to introduce logging mechanisms (Narayanasamy, Pokam, and Calder 2005; Jaygarl et al. 2010; Artzi, Kim, and Ernst 2008), which is known to slow down the subject system. In addition, recording system behavior by means of instrumentation may yield privacy concerns. Tools and approaches that only use data about a crash – such as core dump or exception stack crashes – face a different set of limitations. They have to reconstruct the timeline of

events that have led to the crash (N. Chen 2013b; Mathieu Nayrolles, Hamou-Lhadj, et al. 2015). Computing all the paths from the initial state of the software to the crash point is an NP-complete problem, and may cause state space explosion (N. Chen 2013b; Clause and Orso 2007).

In order to overcome these limitations, some researchers have proposed to use various SMT (satisfiability modulo theories) solvers (Dutertre and Moura 2006) and model checking techniques (Visser et al. 2003). However, these techniques require knowledge that goes beyond traditional software engineering, which hinders their adoption (Visser, Psreanu, and Khurshid 2004).

It is worth mentioning that both categories share a common limitation. It is possible for the required condition to reproduce a crash to be purely external such as the reading of a file that is only present on the hard drive of the customer or the reception of a faulty network packet (N. Chen 2013b; Mathieu Nayrolles, Hamou-Lhadj, et al. 2015). It is almost impossible to reproduce the bug without this input.

3.5.1 On-field Record and In-house Replay

Jaygarl et al. created OCAT (Object Capture based Automated Testing) (Jaygarl et al. 2010). The authors' approach starts by capturing objects created by the program when it runs on-field in order to provide them with an automated test process. The coverage of automated tests is often low due to lack of correctly constructed objects. Also, the objects can be mutated by means of evolutionary algorithms. These mutations target primitive fields in order to create even more objects and, therefore, improve the code coverage. While not directly targeting the reproduction of a bug, OCAT is an approach that was used as the main mechanism for bug reproduction systems.

Narayanasamy et al. (Narayanasamy, Pokam, and Calder 2005) proposed BugNet, a tool that continuously records program execution for deterministic replay debugging. According to the authors, the size of the recorded data needed to reproduce a bug with high accuracy is around 10MB. This recording is then sent to the developers and allows the deterministic replay of a bug. The authors argued that with nowadays Internet bandwidth the size of the recording is not an issue during the transmission of the recorded data.

Another approach in this category was proposed by Clause et al. (Clause and

Orso 2007). The approach records the execution of the program on the client side and compresses the generated data. Moreover, the approach keeps compressed traces of all accessed documents in the operating system. This data is sent to the developers to replay the execution of the program in a sandbox, simulating the client’s environment. This special feature of the approach proposed by Clause et al. addresses the limitation where crashes are caused by external causes. While the authors broaden the scope of reproducible bugs, their approach records a lot of data that may be deemed private such as files used for the proper operation of the operating system.

Timelapse (Burg et al. 2013) also addresses the problem of reproducing bugs using external data. The tool focuses on web applications and allows developers to browse and visualize the execution traces recorded by Dolos. Dolos captures and reuses user inputs and network responses to deterministically replay a field crash. Also, both Timelapse and Dolos allow developers to use conventional tools such as breakpoints and classical debuggers. Similar to the approach proposed by Clause et al. (Clause and Orso 2007), private data are recorded without obfuscation of any sort.

Another approach was proposed by Artzi et al. and named ReCrash. ReCrash records the object states of the targeted programs (Artzi, Kim, and Ernst 2008). The authors use an in-memory stack, which contains every argument and object clone of the real execution in order to reproduce a crash via the automatic generation of unit test cases. Unit test cases are used to provide hints to the developers about the buggy code. This approach particularly suffers from the limitation related to slowing down the execution. The overhead for full monitoring is considerably high (between 13% and 64% in some cases). The authors propose an alternative solution in which they record only the methods surrounding the crash. For this to work, the crash has to occur at least once so they could use the information causing the crash to identify the methods surrounding it. ReCrash was able to reproduce 100% (11/11) of the submitted bugs.

Similar to ReCrash, JRapture (Steven et al. 2000) is a capture/replay tool for observation-based testing. The tool captures the execution of Java programs to replay it in-house. To capture the execution of a Java program, the creators of JRapture used their own version of the Java Virtual Machine (JVM) and a lightweight, transparent capture process. Using a customized JVM allows capturing any interactions between a Java program and the system including GUI, files, and console inputs.

These interactions can be replayed later with exactly the same input sequence as seen during the capture phase. However, using a custom JVM is not a practical solution. This is because the authors’ approach requires users to install a JVM that might have some discrepancies with the original one and yield bugs if used with other software applications. In our view, JRapture fails to address the limitations caused by instrumentation because it imposes the installation of another JVM that can also monitor other software systems than the intended ones. RECORE (REconstructing CORE dumps) is a tool proposed by Robler et al. The tool instruments Java byte code to wrap every method in a try-catch block while keeping a quasi-null overhead (Robler et al. 2013). RECORE starts from the core dump and tries (with evolutionary algorithms) to reproduce the same dump by executing the subject program many times. When the generated dump matches the collected one, the approach has found the set of inputs responsible for the failure and was able to reproduce 85% (6/7) of the submitted bugs.

The approaches presented at this point operate at the code level. There exist also techniques that focus on recording user-GUI interactions (Herbold et al. 2011; Roehm, Nosovic, and Bruegge 2015). Roehm et al. extract the recorded data using delta debugging (Zeller and Hildebrandt 2002), sequential pattern mining, and their combination to reproduce between 75% and 90% of the submitted bugs while pruning 93% of the actions.

Among the approaches presented here, only the ones proposed by Clause et al. and Burg et al. address the limitations incurred due to the need for external data at the cost, however, of privacy. To address the limitations caused by instrumentation, the RECORE approach proposes to let users choose where to put the bar between the speed of the subject program, privacy, and bug reproduction efficiency. As an example, users can choose to contribute or not to improving the software – policy employed by many major players such as Microsoft in Visual Studio or Mozilla in Firefox – and propose different types of monitoring where the cost in terms of speed, privacy leaks, and efficiency for reproducing the bug is clearly explained.

3.5.2 On-house Crash Explanation

On the other side of the picture, we have tools and approaches belonging to the on-house crash explanation (or understanding), which are fewer but newer than on-field

record and replaying tools.

Jin et al. proposed BugRedux for reproducing field failures for in-house debugging (Jin and Orso 2012). The tool aims to synthesize in-house executions that mimic field failures. To do so, the authors use several types of data collected in the field such as stack traces, crash stack at points of failure, and call sequences. The data that successfully reproduced the field crash is sent to software developers to fix the bug. BugRedux relies on several in-house executions that are synthesized so as to narrow down the search scope, find the crash location, and finally reproduce the bug. However, these in-house executions have to be conducted before the work on the bug really begins. Also, the in-house executions suffer from the same limitation as unit testing, i.e., the executions are based on the developer’s knowledge and ability to develop exceptional scenarios in addition to the normal ones. Based on the success of BugRedux, the authors built F3 (Fault localization for Field Failures) (Jin and Orso 2013) and MIMIC (Zuddas et al. 2014). F3 performs many executions of a program on top of BugRedux in order to cover different paths that are leading to the fault. It then generates many *pass* and *fail* paths, which can lead to a better understanding of the bug. They also use grouping, profiling and filtering, to improve the fault localization process. MIMIC further extends F3 by comparing a model of correct behavior to failing executions and identifying violations of the model as potential explanations for failures.

While being close to our approach, BugRedux and F3 may require the call sequence and/or the complete execution trace in order to achieve bug reproduction. When using only the crash traces (referred to as call stack at crash time in their paper), the success rate of BugRedux significantly drops to 37.5% (6/16). The call sequence and the complete execution trace required to reach 100% accuracy can only be obtained through instrumentation and with an overhead ranging from 10% to 1066%. Chronicle (Bell, Sarda, and Kaiser 2013) is an approach that supports remote debugging by capturing inputs in the application through code instrumentation. The approach seems to have a low overhead on the instrumented application, around 10%.

Likewise, Zamfir et al. proposed ESD (Zamfir and Candea 2010), an execution synthesis approach that automatically synthesizes failure execution using only the stack trace information. However, this stack trace is extracted from the core dump

and may not always contain the components that caused the crash.

To the best of our knowledge, the most complete work in this category is the one of Chen in his PhD thesis (N. Chen 2013b). Chen proposed an approach named STAR (Stack Trace based Automatic crash Reproduction). Using only the crash stack, STAR starts from the crash line and goes backward towards the entry point of the program. During the backward process, STAR computes the required condition using an SMT solver named Yices (Dutertre and Moura 2006). The objects that satisfy the required conditions are generated and orchestrated inside a JUnit test case. The test is run, and the resulting crash stack is compared to the original one. If both match, the bug is said to be reproduced. STAR aims to tackle the state explosion problem of reproducing a bug by reconstructing the events in a backward fashion and therefore saving numerous states to explore. STAR was able to reproduce 38 bugs out of 64 (54.6%). Also, STAR is relatively easy to implement as it uses Yices (Dutertre and Moura 2006) and potentially Z3 (De Moura and Bjørner 2008) (stated in their future work) that are well-supported SMT solvers.

Except for STAR, existing approaches that target the reproduction of field crashes require the instrumentation of the code or the running platform in order to save the stack call or the objects to successfully reproduce bugs. As we discussed earlier, such approaches yield good results 37.5% to 100%, but the instrumentation can cause a massive overhead (1% to 1066%) while running the system. In addition, the data generated at run-time using instrumentation may contain sensitive information.

3.6 Bugs Classification

Researchers have been studying the relationships between the bug and source code repositories for more than two decades. To the best of our knowledge, the first ones who conducted this type of study on a significant scale were Perry and Stieg (Perry, Dewayne E. and Stieg. 1993). In these two decades, many aspects of these relationships have been studied in length. For example, researchers were interested in improving the bug reports themselves by proposing guidelines (Bettenburg et al. 2008), and by further simplifying existing bug reporting models (Herraiz et al. 2008).

Another field of study consists of assigning these bug reports, automatically if possible, to the right developers during triaging (Anvik, Hiew, and Murphy 2006;

Jeong, Kim, and Zimmermann 2009; Tamrawi et al. 2011; Bortis and Hoek 2013). Another set of approaches focus on how long it takes to fix a bug (Zhang, Gong, and Versteeg 2013; Bhattacharya and Neamtiu 2011; Saha, Khurshid, and Perry 2014) and where it should be fixed (Zhou, Zhang, and Lo 2012; Zeller 2013). With the rapidly increasing number of bugs, the community was also interested in prioritizing bug reports (Kim et al. 2011), and in predicting the severity of a bug (Lamkanfi et al. 2010). Finally, researchers proposed approaches to predict which bug will get reopened (Zimmermann et al. 2012; Lo 2013), which bug report is a duplicate of another one (Bettenburg, Premraj, and Zimmermann 2008; Tian, Sun, and Lo 2012; Jalbert and Weimer 2008) and which locations are likely to yield new bugs (Kim, Zimmermann, Whitehead Jr., et al. 2007b; Kim, Zimmermann, Pan, et al. 2006a; Tufano et al. 2015). However, to the best of our knowledge, there are not many attempts to classify bugs the way we present in this paper. In her PhD thesis (Eldh 2001), Sigrid Eldh discussed the classification of trouble reports concerning a set of fault classes that she identified. Fault classes include computational logical faults, resource faults, function faults, etc. She conducted studies on Ericsson systems and showed the distributions of trouble reports with respect to these fault classes. A research paper was published on the topic in (Eldh 2001). or safety critical (Hamill and Goseva-Popstojanova 2014). Hamill et al. (Hamill and Goseva-Popstojanova 2014) proposed a classification of faults and failures in critical safety systems. They proposed several types of faults and showed how failures in critical safety systems relate to these classes. They found that only a few fault types were responsible for the majority of failures. They also compare on pre-release and post-release faults and showed that the distributions of fault types differed for pre-release and post-release failures. Another finding is that coding faults are the most predominant ones.

Our study differs from these studies in the way that we focus on the bugs and their fixes across a wide range of systems, programming languages, and purposes. This is done independently from a specific class of faults (such as coding faults, resource faults, etc.). This is because our aim is not to improve testing as it is the case in the work of Eldh (Eldh 2001) and Hamill et al. (Hamill and Goseva-Popstojanova 2014). Our objective is to propose a classification that can allow researchers in the field of mining bug repositories to use the taxonomy as a new criterion in triaging, prediction, and reproduction of bugs. By analogy, we can look at the proposed bug

taxonomy similarly to the clone taxonomy presented by Kapser and Godfrey (Kapser and Godfrey 2003). The authors proposed seven types of source code clones and then conducted a case study, using their classification, on the file system module of the Linux operating system. This clone taxonomy continues to be used by researchers to build better approaches for detecting a given clone type and being able to compare approaches with each other effectively.

Chapter 4

An Aggregated Bug Repository for Developers and Researchers

4.1 Introduction

Program debugging, an important software maintenance activity, is known to be challenging and labor-intensive. Studies have shown that the cost of software maintenance can reach up to 70% of the overall cost of the software development process (Pressman 2005).

When facing a new bug, one might want to leverage decades of open source software history to find a suitable solution. The chances are that a similar bug or crash has already been fixed somewhere in another open source project. The problem is that each open source project hosts its data in a different data repository, using different bug tracking and version control systems. Moreover, these systems have different interfaces to access data.

The data is not represented in a uniform way either. This is further complicated by the fact that bug tracking tools and version control systems are not necessarily connected. The former follows the life of the bug, while the latter manages the fixes. As a general practice, developers create a link between the bug report system and the version control tool by either writing the bug #ID in their commits message or add a link towards the changeset as a comment in the bug report system. As a result, one would have to search the version control system repository to find candidate solutions.

Moreover, developers mainly use classical search engines that index specialized

sites such as StackOverflow¹. These sites are organized in the form of question-response where a developer submits a problem and receives answers from the community. While the answers are often accurate and precise, they do not leverage the history of open source software that has been shown to provide useful insights to help with many maintenance activities such as bug fixing (Saha, Khurshid, and Perry 2014), bug reproduction (Mathieu Nayrolles, Hamou-Lhadj, et al. 2015), fault analysis (Nessa et al. 2008), etc.

In this paper, we introduce BUMPER (BUg Metarepository for dEvelopers and Researchers), a web-based infrastructure that can be used by software developers and researchers to access data from diverse repositories using natural language queries transparently, regardless of where the data was originally created and hosted.

The idea behind BUMPER is that it can connect to any bug tracking and version control systems and download the data into a single database. We created a common schema that represents data, stored in various bug tracking and version control systems. BUMPER uses a web-based interface to allow users to search the aggregated database by expressing queries through a single point of access. This way, users can focus on the analysis itself and not on the way the data is represented or located. BUMPER supports many features including: (1) the ability to use multiple bug tracking and control version systems, (2) the ability to search very efficiently large data repositories using both natural language and a specialized query language, (3) the mapping between the bug reports and the fixes, and (4) the ability to export the search results in Json, CSV and XML formats.

4.2 Approach

4.2.1 Architecture

Figure 6 shows the overall architecture of BUMPER. BUMPER relies on a highly scalable architecture composed of two Virtual Private Servers (VPS), hosted on a physical server.

The first server handles the web requests and runs three distinct components: Pound, Varnish, and NginX. Pound is a lightweight open source reverse proxy program and application firewall. It also serves to decode https request to http. Translating

¹<http://stackoverflow.com/>

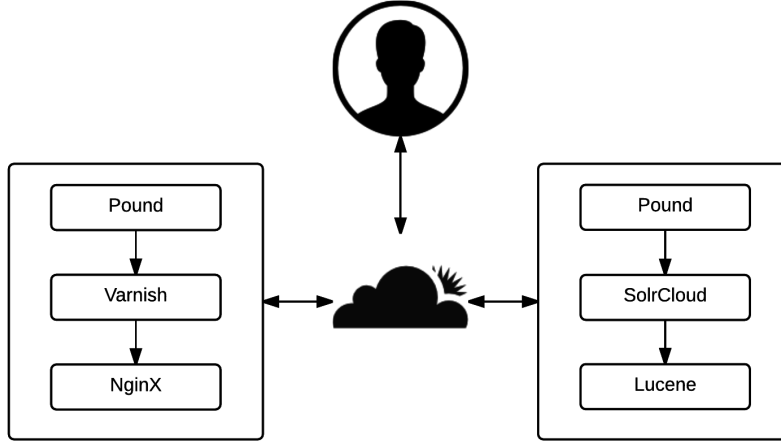


Figure 6: Bumper Architecture

an https request to http allows us to save the https decryption time required on each step. Pound also acts as a load-balancing service for the lower levels. Varnish then handles the translated requests. Varnish is an http accelerator designed for content-heavy and dynamic websites. It caches requests that come in and serves the Web requests from the cache if the cache is still valid. NginX (pronounced engine-x) is a web-server that was developed with a particular focus on high concurrency, high performances, and low memory usage. The second VPS also uses Pound for the same reasons and SolrCloud. SolrCloud is the scalable version of Apache Solr where the data can be separated into shards (e.g., chunks of manageable size). Each shard can be hosted on a different server, but still indexed in a central repository. Hence, we can guarantee a low query time despite the large size of the data. Finally, Lucene is the full text search engine powering Solr. This highly scalable architecture allows BUMPER to serve requests in less than a second in average.

4.2.2 BUMPER Metadata

Figure 7 shows the core BUMPER metamodel, which captures the common data elements used by most existing bug tracking and control version systems. An issue (task) is characterized by a date, a title, a description, and a fixing time.

Issues are reported (created) by and assigned to users. Also, issues belong to a project that is in a repository and might be composed of subprojects. Users can modify an issue during life cycle events which impact the type, the resolution, the

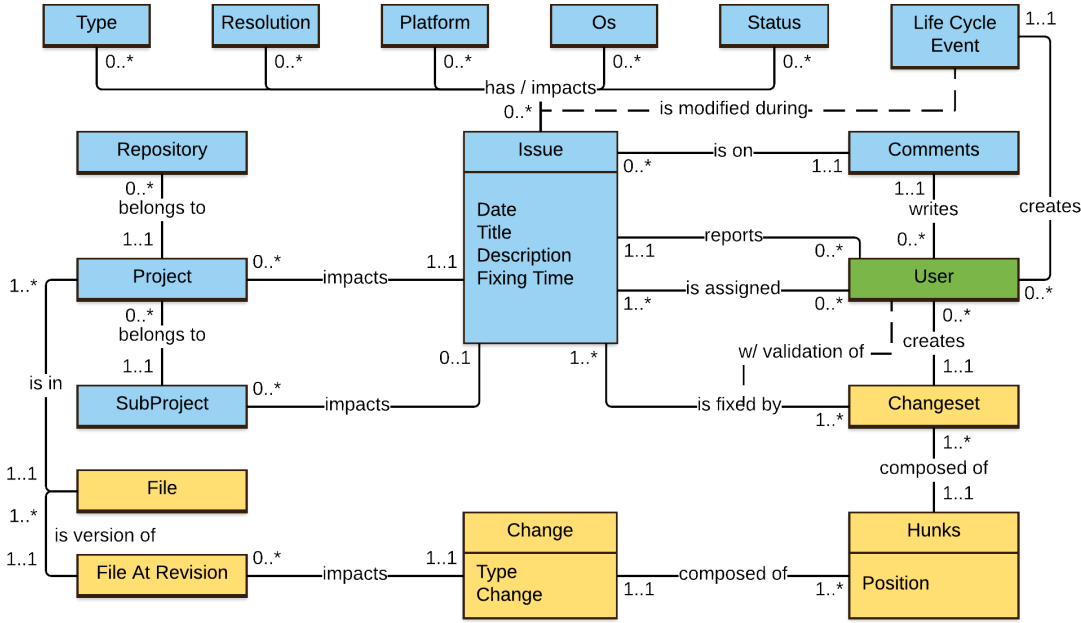


Figure 7: BUMPER Metamodel

platform, the OS and the status. Issues are resolved (implemented) by changesets that are composed of hunks. Hunks contain the actual changes to a file at a given revision, which are versions of the file entity that belongs to a project.

In addition, BUMPER has different indexes over the data in order to provide results efficiently. First of all, each feature of bug reports and bug fixes (number of files, number of hunks, etc.) have their own index. Furthermore, BUMPER has data indexes over the project name, the sub-project name, the programming language, etc. Finally, two major indexes are built upon the concatenations of all the textual fields of the bug report and all the textual field (source code included) of a bug fix. They are named `report_t` and `fix_t`, respectively. These indexes are used when querying bug reports or bug fixes using natural language. Indexes can also be found in relational database management system (RDBMS). Most modern RDBMSs use binary trees to store indexes that keep data sorted and allow searches, insertions, and deletion in logarithmic time. Nevertheless, an RDBMS can use only one index per table at the same time. This said, an RDBMS chooses only one index within the available ones—based on statistics—and uses it to complete the request. It is highly probable that thousands of records have to be scanned before the RDBMS finds the ones that match the query, especially if there is union or disjunction of records.

Unlike a traditional RDBMS, BUMPER relies on Apache Lucene and uses compressed bitsets to store indexes. Bitsets are one of the simplest—and older—data structure that contains only 0 and 1. BUMPER supports binary operations like intersection, AND, OR and XOR that can be performed in a snap even for thousands and thousands of records. As an example, if we wish to retrieve bug reports that contain the words `null pointer exception` and have a changeset containing a `try/catch`, a binary intersection will be performed between the two sets of documents, which is much faster than selecting bug reports that match `null pointer exception` first and then checking if they have a changeset containing a `try/catch` as in the case of an RDBMS. This technique comes with a high overhead—compared to an RDBMS—for index update, but, in practice, information retrievals tend to be much faster. In our case, we want to provide fast access to decades of open source historical data. We periodically update (at night) our indexes when a sufficient amount of new data has been downloaded from the bug tracking and version control systems that BUMPER supports.

4.2.3 Bumper Query Language and API

BUMPER supports two query modes: basic and advanced. The basic query insert users’ inputs (YOUR TERMS) in the following query:

$$\begin{aligned} & (type : "BUG" \text{ AND } report_t : "YOUR TERMS" \\ & \text{ AND } -churns : 0) \end{aligned} \tag{1}$$

The first part of the query matches all bug reports (`type:"BUG"`) that contain YOUR TERMS in the `report.t` index (`report.t:"YOUR TERMS"`). Finally, it excludes bug reports that do not have a fix (`-churns:0`). The result of this subquery will be merged with the following:

$$\begin{aligned} & OR (\{!parent \text{ which} = "type : BUG"\} type : \\ & "CHANGESET" \text{ AND } fix_t : "YOUR TERMS" \end{aligned} \tag{2}$$

This query selects all bugs’ changeset—using a parent-child relationship (`{!parent~
↪ which="type:BUG"} type: "CHANGESET"`) – and that contain YOUR TERMS

in the `fix.t` index (`fix.t:"YOUR TERMS"`). Finally, the results of the two previous queries will be added to the following subquery:

$$\begin{aligned}
 &OR \{ \{!parent\ which = "type : BUG"\} \\
 &\{!parent\ which = "type : CHANGESET"\} \\
 &type : "HUNKS" \ AND\ fix.t : "YOUR TERMS" \}
 \end{aligned} \tag{3}$$

The query selects all the hunks that are children of changsets and grand-children of bug report (`\{!parent\ which="type:BUG"\} \{!parent\ which="type:CHANGESET \rightarrow "\} type:"HUNKS"`) and that contain YOUR TERMS in the `fix.t` index. This composed query intends to search efficiently for YOUR TERMS in bug reports, commit messages and source code all together. The advanced query mode allows users to write their own queries using the indexes they want and the unions or disjunctions they need. As an example, using the advanced query mode, one could write the following:

$$\begin{aligned}
 &(type : "BUG" \ AND\ report_t : "Exception" \\
 &\ AND\ (project : "Axis2" \ OR\ project : "ide") \\
 &\ AND\ (reporter : "Rich" \ OR\ resolution : "fixed") \\
 &\ AND\ (severity : "Major" \ OR\ fixing_time : [10\ TO\ *]) \\
 &\ AND\ -\ churns : 0)
 \end{aligned} \tag{4}$$

This query finds all bug reports that contain Exception in the `report.t` index (first line) and belong to the Axis2 or the ide project (line 2) and have been reported by someone named Rich or have been fixed as a resolution (third line) and that have a Major severity or a fixing time greater than 10 days (fourth line) and have a fix (fifth line).

4.2.4 Bumper Data Repository

Currently, BUMPER supports five bug report management systems, namely, Gnome, Eclipse, Netbeans and the Apache Software Foundation that are composed of 512, 190, 39 and 349 projects respectively, bringing the total of projects supported by BUMPER to 1,930. These projects cover 16 years of development from 1999 to 2015.

Table 1: Resolved/Fixed Bug (R/F BR), Changesets (CS), and projects by dataset

Dataset	R/F BR	CS	Files	Projects
Gnome	550,869	1,231,354	367,245	512
Netbeans	53,258	122,632	30,595	39
Apache	49,449	106,366	38,111	349
Eclipse	78,830	184,900	21,712	190
Total	732,406	1,645,252	457,663	1,930

Gnome is a free desktop environment, mainly developed in C and C++. Eclipse and Netbeans are integrated development environments (IDEs) for developing with many programming languages, including Java, PHP, and C/C++. Finally, The Apache Software Foundation (ASF) is a non-profit public charity established in 1999, that provides services and support for many like-minded software project communities of individuals who choose to join the ASF. The extracted data is consolidated in one database where we associate each bug report with its fix. The fixes are mined from different types of version control systems. Gnome, Eclipse and Apache Software Foundation projects are based on Git (or have git-based mirrors), whereas Netbeans uses Mercurial. The characteristics of the five datasets, aggregated in BUMPER, are presented in Table 1.

As we can see from the table, our consolidated dataset contains 732,406 bugs, 1,645,252 changesets, 457,663 files that have been modified to fix the bugs and 1,930 distinct software projects belonging to four major organizations. We also collected more than two billions of lines of code impacted by the changesets, identified tens of thousands of sub-projects and unique contributors to these bug report systems.

4.3 Experimental Setup

An example of a real-life scenario where BUMPER can be useful would be that of a developer trying to fix a bug. He or she could copy/paste the faulty code, or the obtained error in the search bar of BUMPER. BUMPER will then return (in seconds) every bug report that contains references to the error at hand in the report or in the code developed to fix the bug in our dataset. The developer can then leverage BUMPER’s search results to either find out that someone already fixed the

exact same bug in another project and reuse the fix, analyse similar bugs—and their fixes— to design a solution for the problem, etc.

We conducted a study to assess the effectiveness of BUMPER to help with bug fixing tasks. We asked a group of developers to fill out an online survey² to see how much time it would take to fix the following made-up bug ³:

When I run the CSVReader with a simple main like this:

```
CsvFileUtils csvFileUtils =
    new CsvFileUtils("quebec.txt");

for (int i = 0; i < 2000; i++) {
    printArray(csvFileUtils.readLine());
}
```

I got the following Exception: Exception in thread "main" java.lang.NullPointerException at CsvFileUtils.readLine(CsvFileUtils.java:22) at Main.main(Main.java:11). Can you please fix CsvFileUtils ?

The web interface of bumper, presented in Figure 8, was to be used for this task.

One possible solution to this bug is to apply the following diff to the CsvFileUtils file. Lines preceded by a minus are removed while lines preceded by a plus are added.

```
public String [] readOneLine() throws IOException{

    if(reader == null){
        reader = new BufferedReader(
            new FileReader(file)
        );
    }

-    String [] result = reader.readLine().split(";");
+    String line = null;
+    if((line = reader.readLine()) != null) {
+        return line.split(";");
+    }
```

²<http://goo.gl/forms/RvYFACkl7a>

³<https://github.com/MathieuNls/bumper-csv>

Exception

About 17889 results (1.21 seconds)

LANGUAGES 10 DATASETS 4

DOWNLOAD

▲ Enable collecting thread cpu timestamps by default
https://netbeans.org/bugzilla/show_bug.cgi?id=189621 java, netbeans, profiler

0 Given the fact that thread cpu timestamps are available to JVM (1.6+) on all major operating systems and obtaining them is reasonably quick [1] we can enable collecting them by def more...

▼

▲ unresolved enum with bits info
https://netbeans.org/bugzilla/show_bug.cgi?id=189777 java, netbeans, cnd

0 struct AAA { __extension__ enum int type : 8; } type is not recognized as field of structure more...

▼

▲ Support for generating JAXB annotations in entity classes
https://netbeans.org/bugzilla/show_bug.cgi?id=181161 form, netbeans, webservises

0 Currently, the IDE has a REST from Entity Classes wizard, which generates service and converter classes that expose a set of entities as a service. However, a cleaner approach wou more...

▼

▲ Hints for creating qualifier types
https://netbeans.org/bugzilla/show_bug.cgi?id=187946 xml, netbeans, javaee

0 In an app utilizing CDI it seems common to use custom qualifier types quite a lot. It'd be nice if the IDE offered similar hints for creating them as for normal Java classes. more...

▼

176068:5422427852ea #189621: Enable collecting thread cpu timestamps by default

- lib.profiler.common/src/org/netbeans/lib/profiler/common/integration/IntegrationUtils.java
- lib.profiler.release/remote-pack-defs/README.txt
- lib.profiler.release/remote-pack-defs/build.xml
- lib.profiler.release/remote-pack-defs/calibrate-16.sh
- lib.profiler.release/remote-pack-defs/profile-linux-16.sh
- lib.profiler/src/org/netbeans/lib/profiler/TargetAppRunner.java
- lib.profiler/src/org/netbeans/lib/profiler/global/CalibrationDataFileIO.java
- lib.profiler/src/org/netbeans/lib/profiler/results/cpu/TimingAdjusterOld.java
- profiler.attach/src/org/netbeans/modules/profiler/attach/providers/AbstractIntegrationProvi
- profiler.j2se/src/org/netbeans/modules/profiler/j2se/JavaApplicationIntegrationProvider.jav
- profiler/src/org/netbeans/modules/profiler/NetBeansProfiler.java
- profiler/src/org/netbeans/modules/profiler/actions/AntActions.java
- profiler/src/org/netbeans/modules/profiler/ui/stp/Bundle.properties
- profiler/src/org/netbeans/modules/profiler/ui/stp/DefaultSettingsConfigurator.java

(14) files, (113) insertions, (34) deletions.

Index: lib.profiler.common/src/org/netbeans/lib/profiler/common/integration/IntegrationUtils.java
=====

@@ -131,6 +131,13 @@
}
}

// Returns batch file extension bat / sh according to current /
selected OS
+ public static String getBatchExtensionString(String targetPlatf
orm, String customExt) {
+ if (isWindowsPlatform(targetPlatform)) {
+ return customExt + ".bat"; //NOI18N
+ }
+ return customExt + ".sh"; //NOI18N
+ }
+ }

SHOWCASE

ADVANCED QUERY MODE

API

DATASETS

PUBLICATIONS

ABOUT

Figure 8: Web Interface of BUMPER.

```
-         return result;  
+         return null  
}
```

4.4 Empirical Validation

Participants were asked to find a code snippet that can be slightly modified in order to fix a bug using BUMPER and Google. The goal of this preliminary study was to compare how fast a suitable fix can be found using BUMPER and Google. We send our survey to 20 participants and received eight responses (40%) and asked them to report on their experience in terms of the time taken to find a fix and the number of Web pages that were browsed. Participants reported then when using Google, it took, on average, 6.94 minutes by examining, in average, 7.57 online sources to find a fix. When using BUMPER, however, it only took around 4.5 minutes. The participants only needed to use BUMPER and not other websites. The feedback we

received from the participants was very encouraging. They mainly emphasized the ability of BUMPER to group many repositories in one place, making the search for similar bugs/fixes practical. We intend, in the future, to conduct a large scale (and more formal) experiment with BUMPER for a thorough evaluation of its effectiveness to help software developers in fixing bugs.

4.5 Threats to Validity

The main threat to validity is our somewhat limited validation as we only received eight responses. While the responses from this eight developers are encouraging, a wider human study could invalidate them.

However, we see also BUMPER as an important tool for facilitating research in the area of mining bug repositories. Studying software repositories to gain insights into the quality of the code is a common practice. This task requires time and skills in order to download and link all the pieces of information needed for adequate mining. BUMPER provides a straightforward interface to export bugs and their fixes into CSV, XML and JSON. In short, BUMPER offers a framework that can be used by software practitioners and researchers to analyse (efficiently) bugs and their fixes without having to go from one repository to another, worry about the way data is represented and saved, or create tools for parsing and retrieving various data attributes. We hope that the community contributes by adding more repositories to BUMPER. This way, BUMPER can become a unified environment that can facilitate bug analysis and mining tasks.

4.6 Chapter Summary

In this chapter, we presented a web-based infrastructure, called BUMPER (BUG Metarepository for dEvelopers and Researchers). BUMPER allows natural language searches in bugs reports, commit messages and source code all together while supporting complex queries. Currently, BUMPER is populated with 1,930 projects, more than 732,406 resolved/fixed and with 1,645,252 changesets from Eclipse, Gnome, Netbeans and the Apache Software foundation. The speed of BUMPER allows developers to use it as a way to leverage decades of history scattered over hundreds of software

projects in order to find existing solutions to their problems. There exist tools such as Codemine (Czerwinka et al. 2013) and Boa (Dyer et al. 2013) that enable the mining and reporting of code repositories. These tools, however, will require the download of all the data and process the mining for each installation. To the best of our knowledge, no attempt has been made towards building unified and online datasets, from multiple sources, where researchers and engineers can easily access all the information related to a bug, or a fix in order to leverage decades of historical information. Moreover, the feedback we received from the users of BUMPER in a preliminary study shows that BUMPER holds real potential in becoming a standard search engine for bugs and fixes in multiple repositories.

This online dataset and its APIs were extensively used for the remaining of this thesis.

Chapter 5

Preventing Code Clone Insertion At Commit-Time

5.1 Introduction

Code clones appear when developers reuse code with little to no modification to the original code. Studies have shown that clones can account for up to 50% of code in a given software system (Baker 1995; Ducasse, Rieger, and Demeyer 1999). Although developers often reuse code on purpose (Kim, Sazawal, and Notkin 2005), code cloning is generally considered as a bad practice in software development because clones may introduce bugs (Kapsner and Godfrey 2006; Juergens et al. 2009; Li et al. 2006).

In the last two decades, there have been many studies and tools that aim at detecting clones. They can be grouped into two categories depending on whether they operate locally on a developer’s workstation (e.g., (Zibran and Roy 2012; Sajnani et al. 2016)) or remotely on a server (e.g., (Yamanaka et al. 2012; Zhang et al. 2013)).

Local clone detection approaches are typically implemented as IDE plugins or external tools. IDE-based methods tend to issue many warnings to developers that may interrupt their work, hence hindering their productivity (Ko et al. 2006). Developers may be reluctant to use external tools unless they are involved in a major refactoring effort. These tools cause overhead because of the context switching they provoke (Robertson et al. 2004; Robertson, Lawrance, and Burnett 2006; Beckwith et al. 2006). This problem is somehow similar to the problem of adopting bug identification tools. Studies have shown that these tools are challenging to use because they do

not integrate well with the day-to-day workflow of developers (Lewis et al. 2013; Foss and Murphy 2015; Layman, Williams, and Amant 2007; Ayewah et al. 2007; Johnson et al. 2013). The problem with remote approaches is that the detection occurs too late in the development process. Once the clones reach the central repository, they can be pulled by other members of the development team, further complicating the removal and management of clones.

In this chapter, we present PRECINCT (PREventing Clones INsertion at Commit-Time) that focuses on preventing the insertion of clones at commit-time (i.e., before they reach the central code repository). PRECINCT is a trade-off between local and remote approaches. The approach relies on the use of pre-commit hooks capabilities of modern source code version control systems. A pre-commit hook is a process that one can implement to receive the latest modification to the source code done by a given developer just before the code reaches the central repository. PRECINCT intercepts this modification and analyses its content to see whether a suspicious clone has been introduced or not. A flag is raised if a code fragment is suspected to be a clone of an existing code segment. This said, only a fraction of the code is analysed incrementally, making PRECINCT computationally efficient. In other words, PRECINCT only operates on parts of the program that changed.

To the best of our knowledge, PRECINCT is the first clone detection technique that operates at commit-time. In this study, we focus on Type 3 clones as they are more challenging to detect (Bellon et al. 2007; Kontogiannis 1997; Kapser and Godfrey 2004). Since Type 3 clones include Type 1 and 2 clones, then these types could be detected by PRECINCT as well. We evaluated the effectiveness of PRECINCT on three systems, written in C and Java. The results show that PRECINCT prevents Type 3 clones from reaching the final source code repository with an average accuracy of 97.7%.

5.2 Approach

The PRECINCT approach is composed of six steps. The first and last steps are typical steps that a developer would do when committing code. The first step is the commit step where developers send their latest changes to the central repository, and the last step is the reception of the commit by the central repository. The

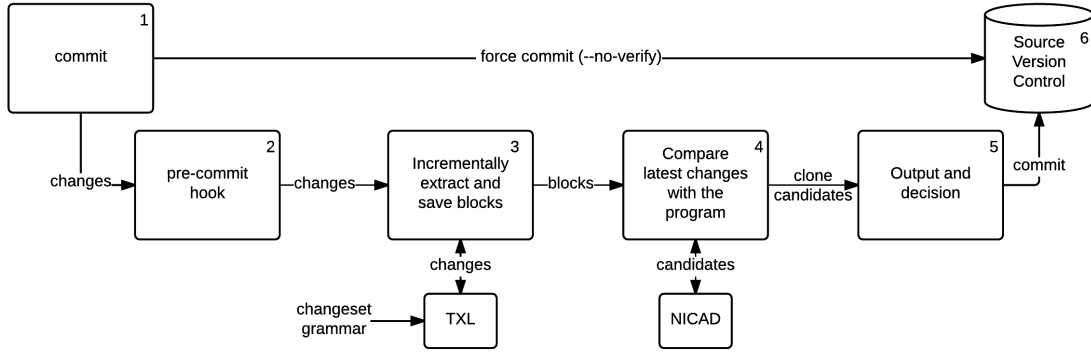


Figure 9: Overview of the PRECINCT approach.

second step is the pre-commit hook, which kicks in as the first operation when one wants to commit. The pre-commit hook has access to the changes regarding the files that have been modified, more specifically, the lines that have been modified. The modified lines of the files are sent to TXL (Cordy 2006) for block extraction. Then, the blocks are compared to previously extracted blocks to identify candidate clones using the comparison engine of NICAD (Cordy and Roy 2011). We chose NICAD engine because it has been shown to provide high accuracy (Cordy and Roy 2011). The tool is also readily available, easy to use, customizable, and works with TXL. Note, however, that PRECINCT can also work with other engines for comparing code fragments if need be. Finally, the output of NICAD is further refined and presented to the user for decision. These steps are discussed in more detail in the following subsections.

5.2.1 Commit

In version control systems, a commit adds the latest changes made to the source code to the repository, making these changes part of the head revision of the repository. Commits in version control systems are kept in the repository indefinitely. Thus, when other users do an update or a checkout from the repository, they will receive the latest committed version, unless they wish to retrieve a previous version of the source code in the repository. Version control systems allow rolling back to previous versions easily. Commits contain bite-sized units of work that are potentially ready to be shared with the rest of the organisation (O’Sullivan and Bryan 2009).

5.2.2 Pre-Commit Hook

Hooks are custom scripts set to fire off when critical actions occur. There are two groups of hooks: client-side and server-side. Client-side hooks are triggered by operations such as committing and merging, whereas server-side hooks run on network operations such as receiving pushed commits. These hooks can be used for all sorts of reasons such as compliance with coding rules or automatic run of unit test suites.

The pre-commit hook is run first, before one even types in a commit message. It is used to inspect the snapshot that is about to be committed. Depending on the exit status of the hook, the commit will be aborted and not pushed to the central repository. Also, developers can choose to ignore the pre-hook. In Git, for example, they will need to use the command `git commit no --verify` instead of `git commit`. This can be useful in case of an urgent need for fixing a bug where the code has to reach the central repository as quickly as possible. Developers can do things like check for code style, check for trailing white spaces (the default hook does exactly this), or check for appropriate documentation on new methods.

PRECINCT is a set of bash scripts where the entry point of these scripts lies in the pre-commit hooks. Pre-commit hooks are easy to create and implement. Note that even though we use Git as the main version control to present PRECINCT, the techniques presented in this paper are readily applicable to other version control systems.

5.2.3 Extract and Save Blocks

A block is a set of consecutive lines of code that will be compared to all other blocks to identify clones. To achieve this critical part of PRECINCT, we rely on TXL (Cordy 2006), which is a first-order functional programming over linear term rewriting, developed by Cordy et al. (Cordy 2006). For TXL to work, one has to write a grammar describing the syntax of the source language and the transformations needed. TXL has three main phases: *parse*, *transform*, *unparse*. In the parse phase, the grammar controls not only the input but also the output form. The code sample below — extracted from the official documentation — shows a grammar matching an *if-then-else* statement in C with some special keywords: [IN] (indent), [EX] (exdent) and [NL] (newline) that will be used for the output form.

```

define if_statement
  if ( [expr] ) [IN][NL]
    [statement] [EX]
    [opt else_statement]
end define

define else_statement
  else [IN][NL]
    [statement] [EX]
end define

```

Then, the *transform* phase will, as the name suggests, apply transformation rules that can, for example, normalize or abstract the source code. Finally, the third phase of TXL called *unparse*, unparses the transformed parsed input to output it. Also, TXL supports what the creators call *Agile Parsing* (Dean et al. 2003), which allows developers to redefine the rules of the grammar and, therefore, apply different rules than the original ones.

PRECINCT takes advantage of that by redefining the blocks that should be extracted for the purpose of clone comparison, leaving out the blocks that are out of scope. More precisely, before each commit, we only extract the blocks belonging to the modified parts of the source code. Hence, we only process, in an incremental manner, the latest modification of the source code instead of the source code as a whole.

We have selected TXL for several reasons. First, TXL is easy to install and to integrate with the standard workflow of a developer. Second, it was relatively easy to create a grammar that accepts commits as input. This is because TXL is shipped with C, Java, C-sharp, Python and WSDL grammars that define all the particularities of these languages, with the ability to customize these grammars to accept changesets (chunks of the modified source code that include the added, modified, and deleted lines) instead of the whole code.

Algorithm 1 presents an overview of the “extract” and “save” blocks operations of PRECINCT. This algorithm receives as arguments, the changesets, the blocks that have been previously extracted and a boolean named `compare_history`. Then, from Lines 1 to 9 lie the *for* loop that iterates over the changesets. For each changeset

(Line 1), we extract the blocks by calling the `extract_blocks(Changeset cs)` function. In this function, we expand our changeset to the left and the right to have a complete block.

```
@@ -315,36 +315,6 @@
int initprocesstree_sysdep
    (ProcessTree_T **reference) {
    mach_port_deallocate(mytask, task);
}
}
- if (task_for_pid(mytask, pt[i].pid,
- &task) == KERN_SUCCESS) {
-     mach_msg_type_number_t    count;
-     task_basic_info_data_t    taskinfo;
```

As depicted by above, changesets contain only the modified chunk of code and not necessarily complete blocks. Indeed, we have a block from Line 2 to Line 6 and deleted lines from Line 7 to 10. However, in Line 7 we can see the end of a block, but we do not have its beginning. Therefore, we need to expand the changeset to the left to have syntactically correct blocks. We do so by checking the block's beginning and ending (using `{` and `}`) in C for example. Then, we send these expanded changesets to TXL for block extraction and formalization.

For each extracted block, we check if the current block overrides (replaces) a previous block (Line 4). In such a case, we delete the previous block as it does not represent the current version of the program anymore (Line 5). Also, we have an optional step in PRECINCT defined in Line 4. The `compare_history` is a condition to delete overridden blocks.

We believe that deleted blocks have been removed for a good reason (bug, default, removed features, ...) and if a newly inserted block matches an old one, it could be worth knowing to improve the quality of the system at hand.

In summary, this step receives the files and lines, modified by the latest changes made by the developer and produces an up to date block representation of the system at hand in an incremental way. The blocks are analyzed in the next step to discover potential clones.

Data: *Changeset*[] changesets;

Block[] prior_blocks;

Boolean compare_history;

Result: Up to date blocks of the systems

for $i \leftarrow 0$ **to** *size_of changesets* **do**

Block[] blocks \leftarrow *extract_blocks(changesets)*;

for $j \leftarrow 0$ **to** *size_of blocks* **do**

if *not compare_history AND blocks[j] overrides one of prior_blocks*

then

 delete *prior_block*;

end

 write *blocks[j]*;

end

end

Function *extract_blocks(Changeset cs)*

if *cs is unbalanced right* **then**

$cs \leftarrow$ *expand_left(cs)*;

else if *cs is unbalanced left* **then**

$cs \leftarrow$ *expand_right(cs)*;

end

1 **return** *txl_extract_blocks(cs)*;

Algorithm 1: Overview of the Extract Blocks Operation

5.2.4 Compare Extracted Blocks

To compare the extracted blocks and detect potential clones, we can only resort to text-based techniques. This is because lexical and syntactic analysis approaches (alternatives to text-based comparisons) would require a complete program to work, a program that compiles. In the relatively wide-range of tools and techniques that exist to detect clones by considering code as text (Johnson 1993; Johnson 1994; Marcus and Maletic 2001; Manber 1994; Ducasse, Rieger, and Demeyer 1999; Wettel and Marinescu 2005), we selected NICAD as the main text-based method for comparing clones (Cordy and Roy 2011) for several reasons. First, NICAD is built on top of TXL, which we also used in the previous step. Second, NICAD can detect Type 1, 2 and 3 clones.

NICAD works in three phases: *Extraction*, *Comparison* and *Reporting*. During the *Extraction* phase all potential clones are identified, pretty-printed, and extracted. We do not use the *Extraction* phase of NICAD as it has been built to work on programs that are syntactically correct, which is not the case for changesets. We replaced NICAD’s *Extraction* phase with our own, described in the previous section.

In the *Comparison* phase, extracted blocks are transformed, clustered and compared to find potential clones. Using TXL sub-programs, blocks go through a process called pretty-printing where they are stripped of formatting and comments. When code fragments are cloned, some comments, indentation or spacing are changed according to the new context where the new code is used. This pretty-printing process ensures that all code will have the same spacing and formatting, which renders the comparison of code fragments easier. Furthermore, in the pretty-printing process, statements can be broken down into several lines. Table 2 (Roy 2009) shows how this can improve the accuracy of clone detection with three for statements, for $(i \rightarrow =0; i<10; i++)$, for $(i=1; i<10; i++)$ and for $(j=2; j<100; j++)$. The pretty-printing allows NICAD to detect Segments 1 and 2 as a clone pair because only the initialization of i is changed. This specific example would not have been marked as a clone by other tools we tested such as Duploc (Ducasse, Rieger, and Demeyer 1999). In addition to the pretty-printing, the code can be normalized and filtered to detect different classes of clones and match user preferences.

Finally, the extracted, pretty-printed, normalized and filtered blocks are marked as potential clones using a Longest Common Subsequence (LCS) algorithm (Hunt and

Table 2: Pretty-Printing Example

Segment 1	Segment 2	Segment 3	S1 & S2	S1 & S3	S2 & S3
for (for (for (1	1	1
i = 0;	i = 1;	j = 2;	0	0	0
i >10;	i >10;	j >100;	1	0	0
i++)	i++)	j++)	1	0	0
Total Matches			3	1	1
Total Mismatches			1	3	3

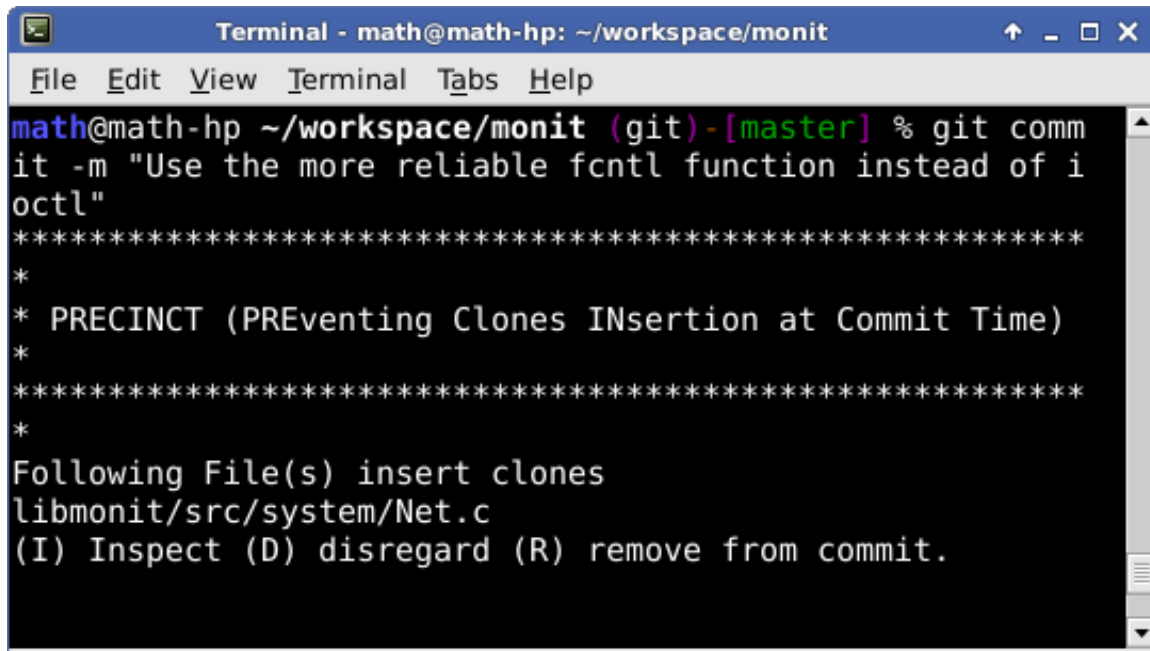
Szymanski 1977). Then, a percentage of unique statements can be computed and, depending on a given threshold (see Section 5.4), the blocks are marked as clones.

The last step of NICAD, which acts as our clone comparison engine, is the *reporting*. However, to prevent PRECINCT from outputting a large amount of data (an issue that many clone detection techniques face), we implemented our reporting system, which is also well embedded with the workflow of developers. This reporting system is the subject of the next section.

As a summary, this step receives potentially expanded and balanced blocks from the extraction step. Then, the blocks are pretty-printed, normalized, filtered and fed to an LCS algorithm to detect potential clones. Moreover, the clone detection in PRECINCT is less intensive than NICAD because we only compare the latest changes with the rest of the program instead of comparing all the blocks with each other.

5.2.5 Output and Decision

In this final step, we report the result of the clone detection at commit-time on the latest changes made by the developer. The process is straightforward. Every change made by the developer goes through the previous steps and is checked for the introduction of potential clones. For each file that is suspected to contain a clone, one line is printed to the command line with the following options: (I) Inspect, (D) Disregard, (R) Remove from the commit. In comparison to this straightforward and interactive output, NICAD outputs each and every detail of the detection result such as the total number of potential clones, the total number of lines, the total number of unique line text chars, the total number of unique lines, and so on. We think that so many details might make it hard for developers to react to these results. A problem that was also raised by Johnson et al. (Johnson et al. 2013) when examining bug

A terminal window titled "Terminal - math@math-hp: ~/workspace/monit" with a menu bar (File, Edit, View, Terminal, Tabs, Help). The prompt is "math@math-hp ~/workspace/monit (git)-[master] %". The command "git commit -m 'Use the more reliable fcntl function instead of ioctl'" has been executed. The output shows a separator of asterisks, followed by the text "* PRECINCT (PREventing Clones INsertion at Commit Time) *", another separator, and then "Following File(s) insert clones" and "libmonit/src/system/Net.c". At the bottom, it lists the options "(I) Inspect (D) disregard (R) remove from commit."

```

Terminal - math@math-hp: ~/workspace/monit
File Edit View Terminal Tabs Help
math@math-hp ~/workspace/monit (git)-[master] % git comm
it -m "Use the more reliable fcntl function instead of i
octl"
*****
*
* PRECINCT (PREventing Clones INsertion at Commit Time)
*
*****
*
Following File(s) insert clones
libmonit/src/system/Net.c
(I) Inspect (D) disregard (R) remove from commit.

```

Figure 10: PRECINCT output when replaying commit 710b6b4 of the Monit system used in the case study.

detection tools. Then the potential clones are stored in XML files that can be viewed using an Internet browser or a text editor.

- (I) Inspect will cause a diff-like visualization of the suspected clones while (D) disregard will simply ignore the finding. To integrate PRECINCT in the workflow of the developer, we also propose the remove option (R). This option will simply remove the suspected file from the commit that is about to be sent to the central repository.

Figure 10 shows an example of PRECINCT’s output when replaying commit 710 → b6b4 of the Monit system. We think that this output, at commit-time, is easy to work with for developers. In addition, developers are used to interacting with their versioning system in the command line. Consequently, we expect a gentle learning curve.

Table 3: List of Target Systems in Terms of Files and Kilo Line of Code (KLoC) at current version and Language

SUT	Revisions	Files	KLoC	Language
Monit	826	264	107	C
Jhotdraw	735	1984	44	Java
dnsjava	1637	233	47	Java

5.3 Experimental Setup

Table 3 shows the systems used in this study and their characteristics in terms of the number files they contain and the size in KLoC (Kilo Lines of Code). We also include the number of revisions used for each system and the programming language in which the system is written.

Monit is a small open source utility for managing and monitoring Unix systems. Monit is used to conduct automatic maintenance and repair and supports the ability to identify causal actions to detect errors. This system is written in C and composed of 826 revisions, 264 files, and the latest version has 107 KLoC. We have chosen Monit as a target system because it was one of the systems NICAD was tested on.

JHotDraw is a Java GUI framework for technical and structured graphics. It has been developed as a “design exercise”. Its design is largely based on the use of design patterns. JHotDraw is composed of 735 revisions, 1984 files, and the latest revision has 44 Kloc. It is written in Java, and researchers often use it as a test bench. JHotDraw was also used by NICAD’s developers to evaluate their approach.

Dnsjava is a tool for implementing the DNS (Domain Name Service) mechanisms in Java. This tool can be used for queries, zone transfers, and dynamic updates. It is not as large as the other two, but it still makes an interesting case subject because it has been well maintained for the past decade. Consequently, it has a large number of revisions. Also, this tool is used in many other popular tools such as Aspirin, Muffin and Scarab. Dnsjava is composed of 1637 revisions, 233 files; the latest revision contains 47 Kloc. We have chosen this system because we are familiar with it as we used it before (Mathieu Nayrolles, Hamou-Lhadj, et al. 2015; Nayrolles et al. 2016).

As our approach relies on commit pre-hooks to detect possible clones during the development process (more particularly at commit-time), we had to find a way to

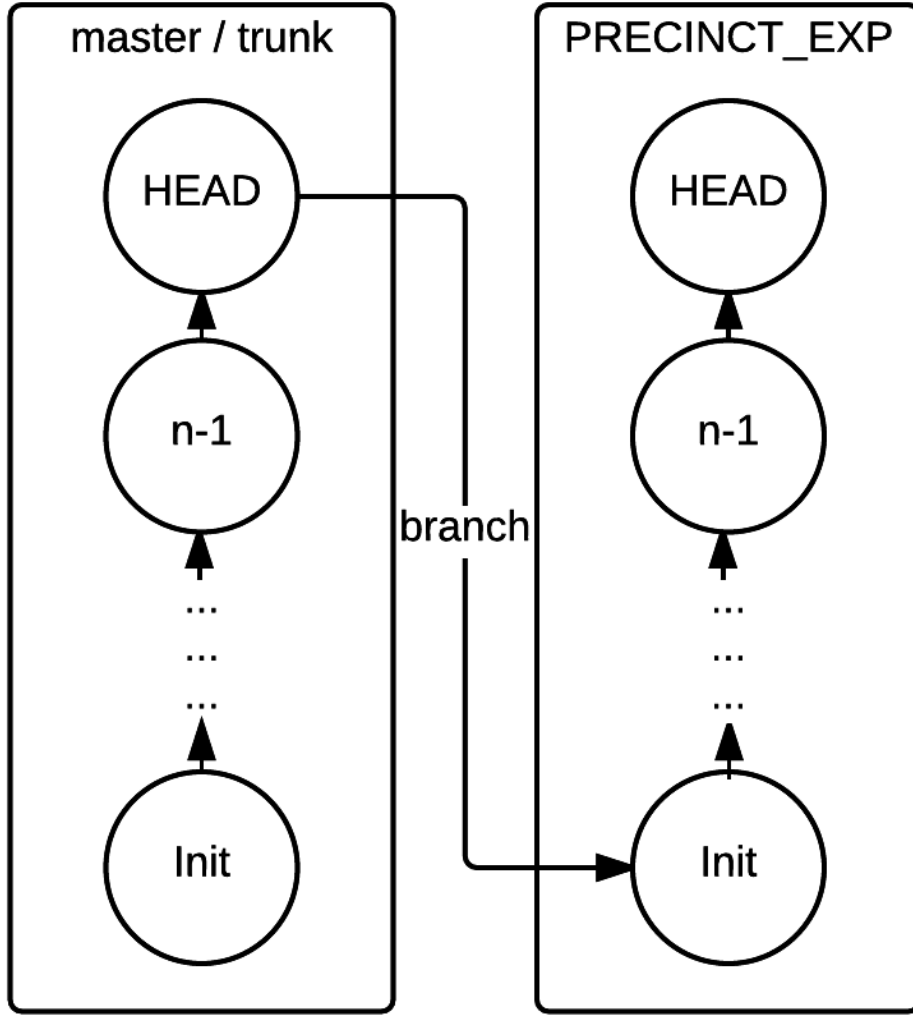


Figure 11: PRECINCT Branching.

replay past commits. To do so, we *cloned* our test subjects, and then created a new branch called *PRECINCT_EXT*. When created, this branch is reinitialized at the initial state of the project (the first commit), and each commit can be replayed as they have originally been. At each commit, we store the time taken for PRECINCT to run as well as the number of detected clone pairs. We also compute the size of the output in terms of the number of lines of text output by our method. The aim is to reduce the output size to help software developers interpret the results.

To validate the results obtained by PRECINCT, we needed to use a reliable clone detection approach to extract clones from the target systems and use these clones as a baseline for comparison. For this, we turned to NICAD because of its popularity,

high accuracy, and availability (Cordy and Roy 2011). This means, we run NICAD on the revisions of the system to obtain the clones then we used NICAD clones as a baseline for comparing the results achieved by PRECINCT.

It may appear strange that we are using NICAD to validate our approach, knowing that our approach uses NICAD’s code comparison engine. In fact, what we are assessing here is the ability for PRECINCT to detect clones at commit-time using changesets. The major part of PRECINCT is the capacity to intercept code changes and build working code blocks that are fed to a code fragment comparison engine (in our case NICAD’s engine). PRECINCT can be based on any other code comparison engine.

We show the result of detecting Type 3 clones with a maximum line difference of 30%. As discussed in the introductory section, we chose to report on Type 3 clones because they are more challenging to detect than Type 1 and 2. PRECINCT detects Type 1 and 2 too, so does NICAD. For the time being, PRECINCT is not designed to detect Type 4 clones. These clones use different implementations. Detecting Type 4 clones is part of future work.

We assess the performance of PRECINCT in terms of precision, recall, and F_1 -measure by using NICAD’s results as ground truth. They are computed using TP (true positives), FP (false positives), FN (false negatives), which are defined as follows:

- TP: is the number of clones that were properly detected by PRECINCT (again, using NICAD’s results as baseline)
- FP: is the number of non-clones that were classified by PRECINCT as clones
- FN: is the number of clones that were not detected by PRECINCT
- Precision: $TP / (TP + FP)$
- Recall: $TP / (TP + FN)$
- F1-measure: $2 \cdot (\text{precision} \cdot \text{recall}) / (\text{precision} + \text{recall})$

5.4 Empirical Validation

Figures 12, 13, 14 show the results of our study in terms of clone pairs that are detected per revision for our three subject systems: Monit, JHotDraw and Dnsjava. We used as a baseline for comparison the clone pairs detected by NICAD. The blue line shows the clone detection performed by NICAD. The red line shows the clone

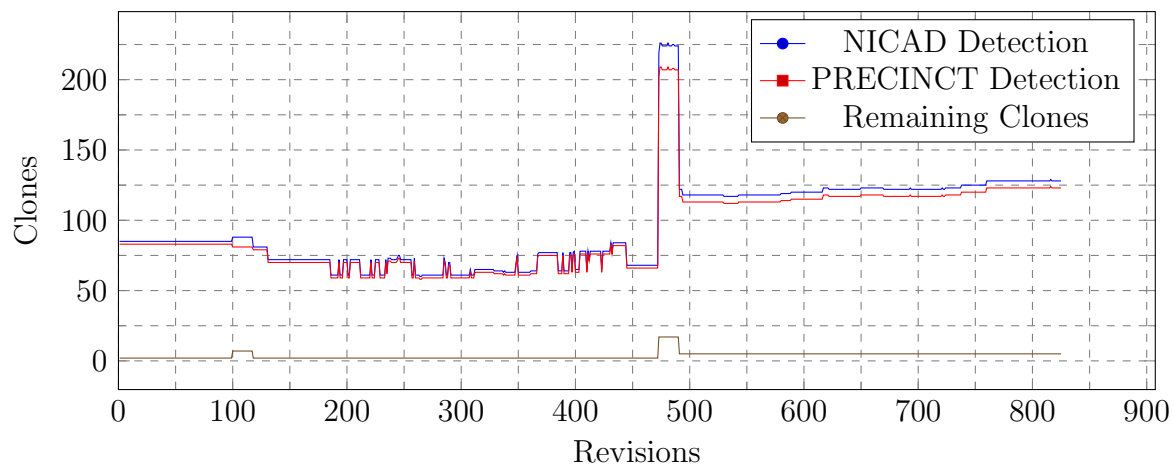


Figure 12: Monit clone detection over revisions

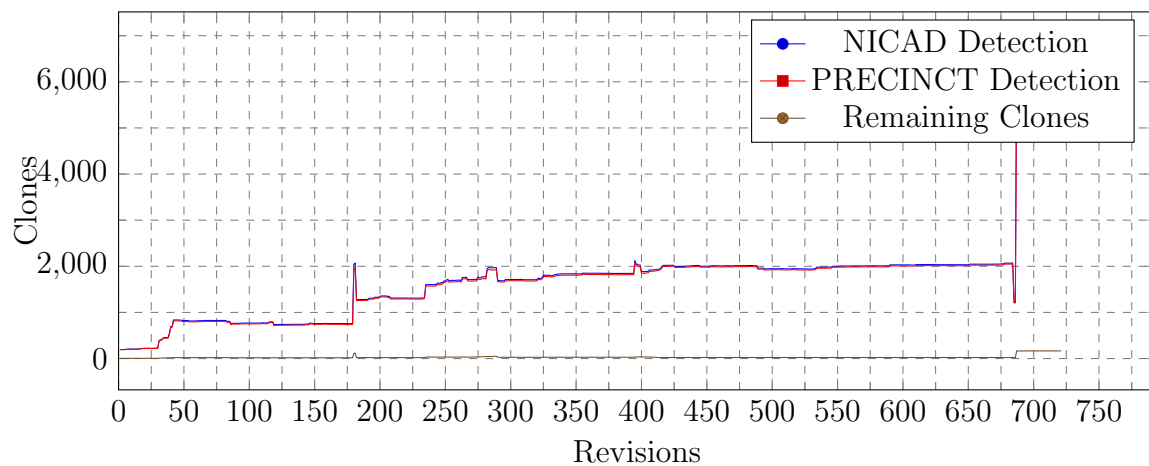


Figure 13: JHotDraw clone detection over revisions

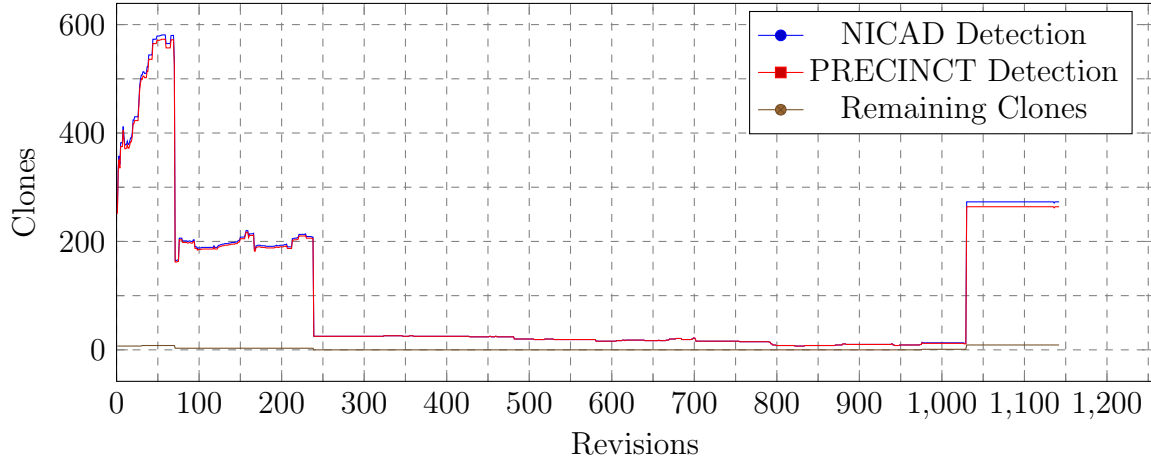


Figure 14: Dnsjava clone detection over revisions

Table 4: Overview of PRECINCT’s results in terms of precision, recall, F_1 -measure, execution time and output reduction.

	Detected	Precision	Recall	F1-measure	NICAD’s Average Execution Time	PRECINCT’s Average Execution Time	
Monit	123	96.1%	100%	98%	2.2s	0.9s	
JHotDraw	6490	98.3%	100%	99.1%	5.1s	1.7s	
DnsJava	226	82.8%	100%	90.6%	1.8s	1.1s	
Total	6839	97.7%	100%	98.8%	3s	1.2s	

pairs detected by PRECINCT. The brown line shows the clone pairs that have been missed by PRECINCT. As we can quickly see, the blue and red lines almost overlap, which indicates a good accuracy of the PRECINCT approach.

Table 4 summarizes PRECINCT’s results in terms of precision, recall, F_1 -measure, execution time and output reduction compared to NICAD for our three subject systems: Monit, JHotDraw, and Dnsjava. The first version of Monit contains 85 clone pairs, and this number stays stable until Revision 100. From Revision 100 to 472 the detected clone pairs vary between 68 and 88 before reaching 219 at Revision 473. The number of clone pairs goes down to 122 at Revision 491 and increases to 128 in the last revision. PRECINCT was able to detect 96.1% (123/128) of the clone pairs that are detected by NICAD with a 100% recall. It took in average around 1 second for PRECINCT to execute on a Debian 8 system with Intel(R) Core(TM) i5-2400 CPU

@ 3.10GHz, 8Gb of DDR3 memory. It is also worth mentioning that the computer we used is equipped with SSD (Solid State Drive). This impacts the running time as clone detection is a file intensive operation. Finally, the PRECINCT was able to output up to 88.3% fewer lines than NICAD.

JHotDraw starts with 196 clone pairs at Revision 1 and reaches a pick of 2048 at Revision 180. The number of clones continues to go up until Revisions 685 and 686 where the number of pairs is 1229 before picking at 6538 from Revisions 687 to 721. PRECINCT was able to detect 98.3% of the clone pairs detected by NICAD (6490/6599) with 100% recall while executing on average in 1.7 seconds (compared to 5.1 seconds for NICAD). With JHotDraw, we can see the advantages of incremental approaches. Indeed, the execution time of PRECINCT is loosely impacted by the number of files inside the system as the blocks are constructed incrementally. Also, we only compare the latest change to the remaining of the program and not all the blocks to each other as NICAD. We also were able to reduce by 70.1% the number of lines outputted.

Finally, for Dnsjava, the number of clone pairs starts high with 258 clones and goes down until Revision 70 where it reaches 165. Another quick drop is observed at Revision 239 where we found only 25 clone pairs. The number of clone pairs stays stable until Revision 1030 where it reaches 273. PRECINCT was able to detect 82.8% of the clone pairs detected by NICAD (226/273) with 100% recall while executing on average in 1.1 seconds while NICAD took 3 seconds in average. PRECINCT outputs 83.4% fewer lines than NICAD.

Overall, PRECINCT prevented 97.7% of the 7000 clones (in all systems) to reach the central source code repository while executing more than twice as fast as NICAD (1.2 seconds compared to 3 seconds in average) and reducing the output in terms of lines of text output to the developers by 83.4% in average. Note here that we have not evaluated the quality of the output of PRECINCT compared to NICAD's output. We need to conduct user studies for this. We are, however, confident, based on our experience trying many clone detection tools, that a simpler and a more interactive way to present the results of a clone detection tool is warranted. PRECINCT aims to do just that.

The difference in execution time between NICAD and PRECINCT stems from the fact that, unlike PRECINCT, NICAD is not an incremental approach. For each

revision, NICAD has to extract all the code blocks and then compares all the pairs with each other. On the other hand, PRECINCT only extracts blocks when they are modified and only compares what has been changed with the rest of the program.

The difference in precision between NICAD and PRECINCT (2.3%) can be explained by the fact that sometimes developers commit code that does not compile. Such commits will still count as a revision, but TXL fails to extract blocks that do not comply with the target language syntax. While NICAD also fails in such a case, the disadvantage of PRECINCT comes from the fact that the failed block is saved and used as a reference until it is changed by a correct one in another commit.

5.5 Threats to Validity

The selection of target systems is one of the common threats to validity for approaches aiming to improve the analysis of software systems. It is possible that the selected programs share common properties that we are not aware of and therefore, invalidate our results. However, the systems analyzed by PRECINCT are the same as the ones used in similar studies. Moreover, the systems vary in terms of purpose, size, and history.

Also, we see a threat to validity that stems from the fact that we only used open source systems. The results may not be generalizable to industrial systems. We intend to undertake these studies in future work.

The programs we used in this study are all based on the Java, C, and Python programming languages. This can limit the generalization of the results. However, similar to Java, C, Python, if one writes a TXL grammar for a new language — which can be relatively hard work — then PRECINCT can work since PRECINCT relies on TXL.

Finally, we use NICAD as the code comparison engine. The accuracy of NICAD affects the accuracy of PRECINCT. This said, since NICAD has been tested on large systems, we are confident that it is a suitable engine for comparing code using TXL. Also, there is nothing that prevents us from using other code comparisons engines, if need be.

In conclusion, internal and external validity have both been minimized by choosing

a set of three different systems, using input data that can be found in any programming languages and version systems (commit and changesets).

5.6 Chapter Summary

In this chapter, we presented PRECINCT (PREventing Clones INsertion at Commit-Time), an incremental approach for preventing clone insertion at commit-time that combines efficient block extraction and clone detection. The approach is meant to integrate well with the workflow of developers. PRECINCT takes advantage of TXL and NICAD to create a clone detection tool that runs automatically before each commit in an average time of 1.2 seconds with an average 97.7% precision and 100% recall (when using NICAD’s results as a baseline).

Our approach is an efficient trade-off between local and remote approaches for clone detection, and as such, we believe that it addresses major factors that contribute to the slow adoption of clone detection tools. PRECINCT achieves similar performance than remote approaches, which rely on more computational power, without delaying the detection results in asynchronous email notifications. Moreover, we believe that operating at commit-time makes PRECINCT more appealing to developers. Using PRECINCT, developers do not have to cope with many warnings and the problem of context-switching, which are the main limitations of IDE-based methods and the use of external tools. Finally, our approach can reduce the number of lines output by a traditional clone detection tool such as NICAD by 83.4% while keeping all the necessary information that would allow developers to decide whether the detected clone is, in fact, a clone.

In the remaining of this thesis, we build upon BUMPER and PRECINCT to detect risky commit and propose solutions to improve code at commit-time.

Chapter 6

Preventing Bug Insertion Using Clone Detection At Commit-Time

6.1 Introduction

Research in software maintenance has evolved over the year to include areas like mining bug repositories, bug analytic, and bug prevention and reproduction. The ultimate goal is to develop better techniques and tools to help software developers detect, correct, and prevent bugs effectively and efficiently.

One particular (and growing) line of research focuses on the problem of preventing the introduction of bugs by detecting risky commits (preferably before the commits reach the central repository). Recent approaches (e.g., (Lo 2013; Nam, Pan, and Kim 2013)) rely on training models based on code and process metrics (e.g., code complexity, experience of the developers, etc.) that are used to classify new commits as risky or not. Metrics, however, may vary from one project to another, hindering the reuse of these models. Consequently, these techniques tend to operate within single projects only, despite the fact that many large projects share dependencies, such as the reuse of common libraries. This makes them potentially vulnerable to similar faults. A solution to a bug provided by the developers of one project may help fix a bug that occurs in another (and dependent) project. Moreover, as noted by Lewis *et al.* (Lewis et al. 2013) and Johnson *et al.* (Johnson et al. 2013), techniques based solely on metrics are perceived by developers as black box solutions because they do not provide any insights on the causes of the risky commits or ways for improving

them. As a result, developers are less likely to trust the output of these tools.

In this chapter, we present a novel bug prevention approach at commit-time, called BIANCA (Bug Insertion ANTicipation by Clone Analysis at commit time). BIANCA does not use metrics to assess whether or not an incoming commit is risky. Instead, it relies on code clone detection techniques by extracting code blocks from incoming commits and comparing them to those of known defect-introducing commits.

One particular aspect of BIANCA is its ability to detect risky commits not only by comparing them to commits of a single project but also to those belonging to other projects that share common dependencies. This is important because complex software systems are not designed monolithically. They have dependencies that make them vulnerable to similar faults. For example, Apache BatchEE (The Apache Software Foundation 2015) and GraphWalker (Graphwalker 2016) both depend on JUNG (Java Universal Network/Graph Framework) (Joshua O'Madadhain, Danyel Fisher, Scott White, Padhraic Smyth 2005). BatchEE provides an implementation of the jsr-352 (Batch Applications for the Java Platform) specification (Chris Vignola 2014) while GraphWalker is an open source model-based testing tool for test automation. These two systems are designed for different purposes. BatchEE is used to do batch processing in Java, whereas GraphWalker is used to design unit tests using a graph representation of the code. Nevertheless, because both Apache BatchEE and GraphWalker rely on JUNG, the developers of these projects made similar mistakes when building upon JUNG. Issue #69 and #44 from Apache BatchEE and Graphwaler, respectively, indicate that the developers of these projects made similar mistakes when using the graph visualization component of JUNG. To detect *risky* commits across projects, BIANCA resorts to project dependency analysis. Note that we do not detect only the bugs resulting from library usage but rather leverage the fact that if two systems use the same libraries, then they are likely vulnerable to the same flaws.

Another advantage of BIANCA is that it uses commits that are used to fix previous defect-introducing commits to guide the developers on how to improve risky commits. This way, BIANCA goes one step further than existing techniques by providing developers with a potential fix for their risky commits.

We validated the performance of BIANCA on 42 open source projects, obtained from Github. The examined projects vary in size, domain and popularity. Our findings indicate that BIANCA is able to flag risky commits with an average precision,

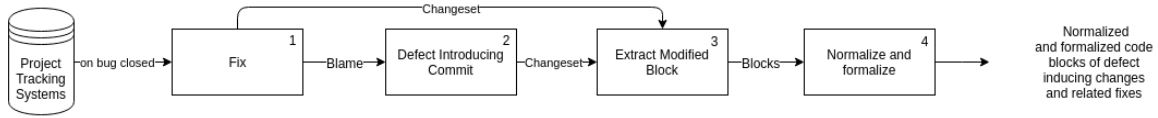


Figure 15: Managing events happening on project tracking systems to extract defect-introducing commits and commits that provided the fixes

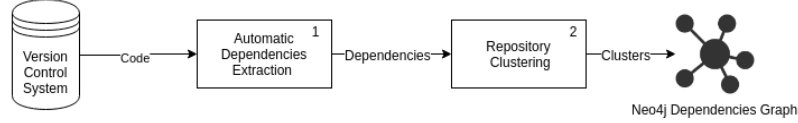


Figure 16: Clustering by dependency

recall and F-measure of 90.75%, 37.15% and 52.72%, respectively. Although the performance of BIANCA seems modest, it is important to ground these results with the fact that BIANCA outperforms a baseline model by 19.48% (mainly due to the data imbalance, i.e., most commits are not defect-inducing). Moreover, we found that only 8.6% of the risky commits detected by BIANCA match other commits from the same project. This finding stresses the fact that relationships across projects should be taken into consideration for effective prevention of risky commits.

6.2 Approach

Figures 15, 16 and 17 show an overview of the BIANCA approach, which consists of two parallel processes.

In the first process (Figures 15 and 16), BIANCA manages events happening on project tracking systems to extract defect-introducing commits and commits that provided the fixes. For simplicity, in the rest of this paper, we refer to commits that are used to fix defects as *fix-commits*. We use the term *defect-commit* to mean a commit that introduces a defect. In the second phase, BIANCA analyses the developer’s new commits before they reach the central repository to detect potential risky commits (commits that may introduce bugs).

The project tracking component of BIANCA listens to bug (or issue) closing events of major open-source projects (currently, BIANCA is tested with 42 large projects). These projects share many dependencies. Projects can depend on each other or

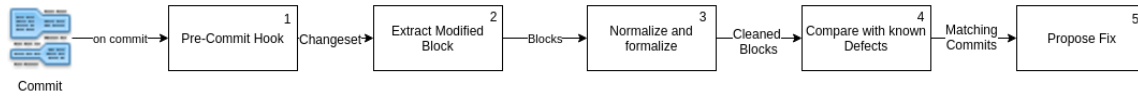


Figure 17: Classifying incoming commits and proposing fixes

common external tools and libraries. We perform project dependency analysis to identify groups of highly-coupled projects.

In the second process (Figure 17), BIANCA identifies risky commits within each group to increase the chances of finding risky commits caused by project dependencies. For each project group, we extract code blocks from defect-commits and fix-commits. The extracted code blocks are saved in a database that is used to identify risky commits before they reach the central repository. For each match between a risky commit and a *defect-commit*, we pull out from the database the corresponding *fix-commit* and present it to the developer as a potential way to improve the commit content. These phases are discussed in more detail in the upcoming subsections.

6.2.1 Clustering Project Repositories

We cluster projects according to their dependencies. The rationale is that projects that share dependencies are most likely to contain defects caused by misuse of these dependencies. In this step, the project dependencies are analysed and saved into a single NoSQL graph database as shown in Figure 16. Graph databases use graph structures as a way to store and query information. In our case, a node corresponds to a project that is connected to other projects on which it depends. Project dependencies can be automatically retrieved if projects use a dependency manager such as Maven.

Figure 31 shows a simplified view of a dependency graph for a project named `com.badlogicgames.gdx`. As we can see, `com.badlogicgames.gdx` depends on projects owned by the same organization (i.e., badlogicgames) and other organizations such as Google, Apple, and Github.

Once the project dependency graph is extracted, we use a clustering algorithm to partition the graph. To this end, we choose the Girvan–Newman algorithm (Girvan and Newman 2002; Newman and Girvan 2004), used to detect communities by progressively removing edges from the original network. Instead of trying to construct

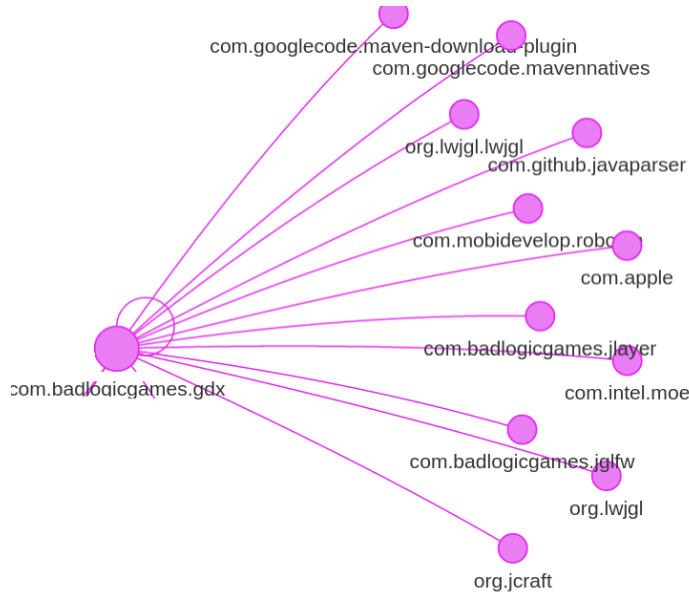


Figure 18: Simplified Dependency Graph for `com.badlogicgames.gdx`

a measure that identifies the edges that are the most central to communities, the Girvan–Newman algorithm focuses on edges that are most likely “between” communities. This algorithm is very effective at discovering community structure in both computer-generated and real-world network data (Newman and Girvan 2004). Other clustering algorithms can also be used.

6.2.2 Building a Database of Code Blocks of Defect-Commits and Fix-Commits

To build our database of code blocks that are related to defect-commits and fix-commits, we first need to identify the respective commits. Then, we extract the relevant blocks of code from the commits.

Extracting Commits: BIANCA listens to bug (or issue) closing events happening in the project tracking system. Every time an issue is closed, BIANCA retrieves the commit that was used to fix the issue (the fix-commit) as well as the one that introduced the defect (the defect-commit). Retrieving fix-commits, however, is known to be a challenging task (Wu et al. 2011). This is because the link between the project tracking system and the code version control system is not always explicit. In an ideal situation, developers would add a reference to the issue they work on inside

the description of the commit. However, this good practice is not always followed. To link fix-commits and their related issues, we turn to a modified version of the back-end of commit-guru (Rosen, Grawi, and Shihab 2015). Commit-guru is a tool, developed by Rosen *et al.* (Rosen, Grawi, and Shihab 2015) to detect *risky commits*. In order to identify risky commits, Commit-guru builds a statistical model using change metrics (i.e., amount of lines added, amount of lines deleted, amount of files modified, etc.) from past commits known to have introduced defects in the past.

Commit-guru’s back-end has three major components: ingestion, analysis, and prediction. We reuse the ingestion and the analysis part for BIANCA. The ingestion component is responsible for ingesting (i.e., downloading) a given repository. Once the repository is entirely downloaded on a local server, each commit is analysed. Commits are classified using the list of keywords proposed by Hindle *et al.* (Hindle, German, and Holt 2008). Commit-guru implements the SZZ algorithm (Kim, Zimmermann, Pan, et al. 2006b) to detect risky changes, where it performs the SCM blame/annotate function on all the modified lines of code for their corresponding files on the fix-commit’s parents. This returns the commits that previously modified these lines of code and are flagged as the defect introducing commits (i.e., the defect-commits). Prior work showed that commit-guru is effective in identifying defect-commits and their corresponding fixing commits (Kamei et al. 2013) and the SZZ algorithm, used by commit-guru, is shown to be effective in detecting risky commits (Kamei et al. 2013; Rosen, Grawi, and Shihab 2015). Note that we could use a simpler and more established approach such as Relink (Wu et al. 2011) to link the commits to their issues and re-implement the classification proposed by Hindle *et al.* (Hindle, German, and Holt 2008) on top of it. However, commit-guru has the advantage of being open-source, making it possible to modify it to fit our needs by fine-tuning its performance.

Extracting Code Blocks: To extract code blocks from fix-commits and defect-commits, we rely on PRECINCT which has been presented in the previous chapter.

6.2.3 Analysing New Commits Using Pre-Commit Hooks

Each time a developer makes a commit, BIANCA intercepts it using a pre-commit hook extracts the corresponding code block (in a similar way as in the previous phase), and compares it to the code blocks of historical defect-commits. If there is a match, then the new commit is deemed to be risky. A threshold α is used to assess the extent

beyond which two commits are considered similar. The setting of α is discussed in the case study section.

BIANCA is based on a set of bash and python scripts, and the entry point of these scripts lies in a pre-commit hook. These scripts intercept the commit and extract the corresponding code blocks.

We use pre-commit hook, which kicks in as the first operation when one wants to commit. The pre-commit hook has access to the changes regarding the files that have been modified, more specifically, the lines that have been modified. The modified lines of the files are sent to TXL with our special grammar and algorithm (Algorithm 1 presented in the previous chapter.

Then, the blocks are compared to previously extracted blocks known to introduce a defect using the comparison engine of NICAD (Cordy and Roy 2011).

To compare the extracted blocks to the ones in the database, we use a similar strategy as the one presented in the previous chapter.

Another important aspect of the design of BIANCA is the ability to provide guidance to developers on how to improve risky commits. We achieve this by extracting from the database the fix-commit corresponding to the matching defect-commit and present it to the developer. We believe that this makes BIANCA a practical approach for the developers as they will know why a given modification has been reported as risky in terms of code; this is something that is not supported by techniques based on statistical models (e.g., (Kamei et al. 2013; Rosen, Grawi, and Shihab 2015)). A tool that supports BIANCA should have enough flexibility to allow developers to enable or disable the recommendations made by BIANCA. Furthermore, because BIANCA acts before the commit reach the central repository, it prevents unfortunate pulls of defects by other members of the organization.

6.3 Experimental Setup

In this section, we present the setup of our case study in terms of repository selection, dependency analysis, comparison process and evaluation measures.

6.3.1 Project Repository Selection

To select the projects used to evaluate our approach, we followed three simple criteria. First, the projects need to be in Java and use Maven to manage dependencies. This way, we can automatically extract the dependencies and perform the clustering of projects. The second criterion is to have projects that enjoy a large community support and interest. We selected projects that have at least 2000 followers. Finally, the projects must have a public issue repository to be able to mine their past issues and the fixes. We queried Github with these criteria and retrieved 42 projects (see Table 6 for the list of projects), including those from some of major open-source contributors such as Alibaba, Apache Software Foundation, Eclipse, Facebook, Google and Square.

6.3.2 Project Dependency Analysis

Figure 30 shows the project dependency graph. The dependency graph is composed of 592 nodes divided into five clusters shown in yellow, red, green, purple and blue. The size of the nodes in Figure 30 is proportional to the number of connections from and to the other nodes.

As shown in Figure 30, these Github projects are very much interconnected. On average, the projects composing our dataset have 77 dependencies. Among the 77 dependencies, on average, 62 dependencies are shared with at least one other project from our dataset.

Table 5 shows the result of the Girvan–Newman clustering algorithm in terms of centroids and betweenness. Storm dominates the blue cluster from The Apache Software Foundation. Storm is a distributed real-time computation system. Druid by Alibaba, the e-commerce company that provides consumer-to-consumer, business-to-consumer and business-to-business sales services via web portals, dominates the yellow cluster. In recent years, Alibaba has become an active member of the open-source community by making some of its projects publicly available. The red cluster has Hadoop by the Apache Software Foundation as its centroid. Hadoop is an open-source software framework for distributed storage and distributed processing of very large datasets on computer clusters built from commodity hardware. The Persistence project of OpenHab dominates the green cluster. OpenHab proposes home automation solutions, and the *persistence* project is their data access layer. Finally, the purple cluster is dominated by Libdx by Badlogicgames, which is a cross-platform

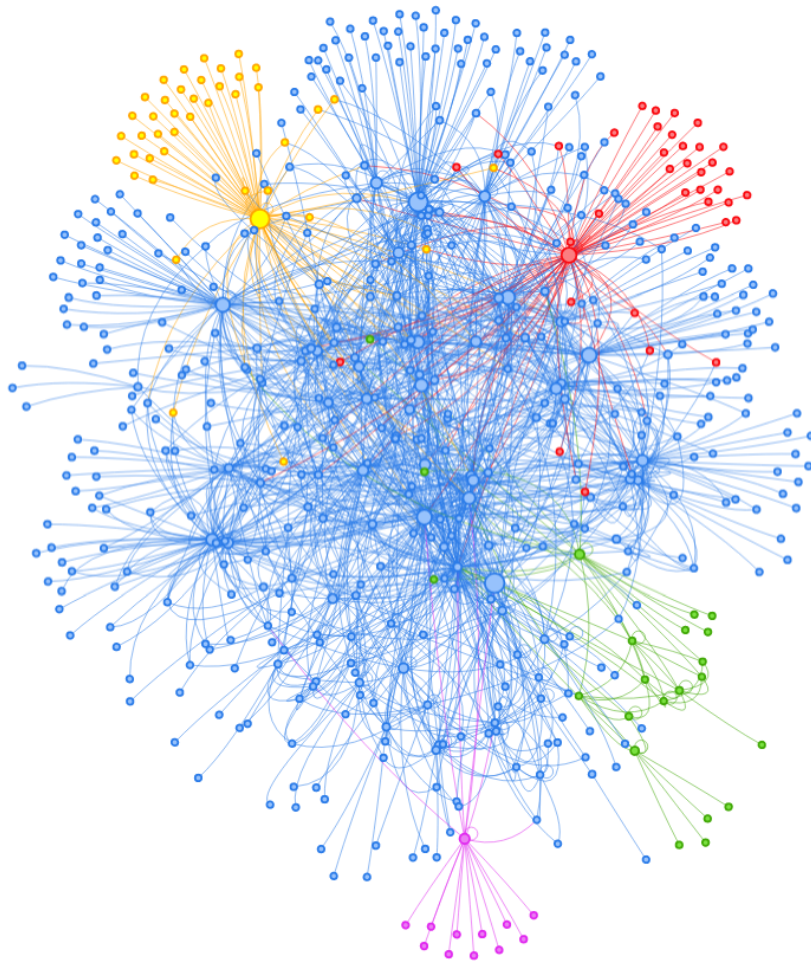


Figure 19: Dependency Graph

Table 5: Communities in terms of ID, Color code, Centroids, Betweenness and number of members

#ID	Community	Centroids	Betweenness	# Members
1	Blue	Storm	24,525	479
2	Yellow	Alibaba	24,400	42
3	Red	Hadoop	16,709	37
4	Green	Openhab	3,504	22
5	Purple	Libdx	6,839	12

framework for game development.

A review of each cluster shows that this partitioning divides projects in terms of high-level functionalities. For example, the blue cluster is almost entirely composed of projects from the Apache Software Foundation. Projects from the Apache Software Foundation tend to build on top of one another. We also have the red cluster for Hadoop, which is by itself an ecosystem inside the Apache Software Foundation. Finally, we obtained a cluster for e-commerce applications (yellow), real-time network application for home automation (green), and game development (purple).

6.3.3 Building a Database of Defect-Commits and Fix-Commits for Performances Evaluation

To build the database that we can use to assess the performance of BIANCA, we use the same process as discussed in Section 7.2.2. We used Commit-guru to retrieve the complete history of each project and label commits as defect-commits if they appear to be linked to a closed issue. The process used by Commit-guru to identify commits that introduce a defect is simple and reliable in terms of accuracy and computation time (Kamei et al. 2013). We use the commit-guru labels as the baseline to compute the precision and recall of BIANCA. Each time BIANCA classifies a commit as *risky*, we can check if the *risky* commit is in the database of defect-introducing commits. The same evaluation process is used by related studies (El Emam, Melo, and Machado 2001; Lee et al. 2011; Bhattacharya and Neamtiu 2011; Kpodjedo et al. 2010).

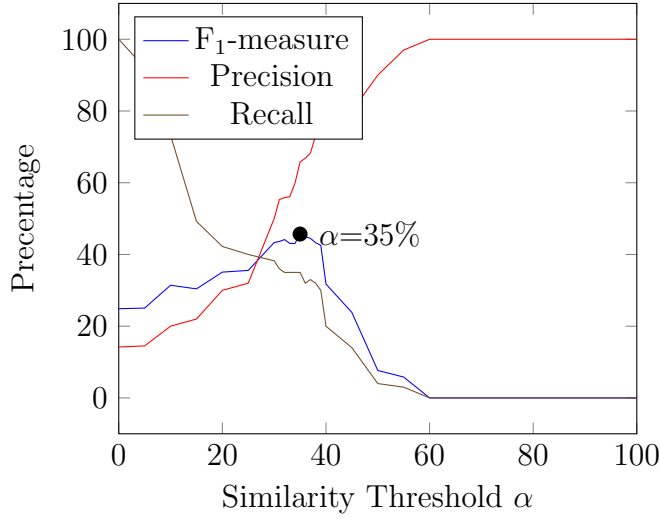


Figure 20: Precision, Recall and F₁-measure variations according to α

6.3.4 Process of Comparing New Commits

Because our approach relies on commit pre-hooks to detect risky commits, we had to find a way to *replay* past commits. To do so, we *cloned* our test subjects, and then created a new branch called *BIANCA*. When created, this branch is reinitialized at the initial state of the project (the first commit) and each commit can be replayed as they have originally been. For each commit, we store the time taken for *BIANCA* to run, the number of detected clone pairs, and the commits that match the current commit. As an example, let's assume that we have three commits from two projects. At time t_1 , commit c_1 in project p_1 introduces a defect. The defect is experienced by a user that reports it via an issue i_1 at t_2 . A developer fixes the defect introduced by c_1 in commit c_2 and closes i_1 at t_3 . From t_3 we know that c_1 introduced a defect using the process described in Section 7.3.3. If at t_4 , c_3 is pushed to p_2 and c_3 matches c_1 after preprocessing, pretty-printing and formatting, then c_3 is classified as *risky* by *BIANCA* and c_2 is proposed to the developer as a potential solution for the defect introduced in c_3 .

To measure the similarity between pairs of commits, we need to decide on the value of α . One possibility would be to test for all possible values of α and pick the one that provides the best accuracy (F₁-measure). The ROC (Receiver Operating Characteristic) curve can then be used to display the performance of *BIANCA* with different values of α . Running experiments with all possible α turned out to be

computationally demanding given the large number of commits. Testing with all the different values of α amounts to 4e10 comparisons.

To address this, we randomly selected a sample of 1700 commits from our dataset and checked the results by varying α from 1 to 100%. Figure 20 shows the results. The best trade-off between precision and recall is obtained when $\alpha = 35\%$.

This threshold is not in line with the findings of Roy et al. (Roy and Cordy 2008; Cordy and Roy 2011) who showed through empirical studies that using NICAD with a threshold of around 70%.

However, Nicad was built for and evaluated on its capacity to detect near-miss clones (i.e. clones that are almost identical). For this purpose, a very-high similarity threshold (or low dissimilarity threshold) is desirable. For our purpose, however, one similar line in a block could be significant. The trivial example being the null check as exposed in the next section. The same reasoning applies for fixes. For these reasons, we set $\alpha = 35\%$ in our experiments.

6.3.5 Evaluation Measures

Similar to prior work focusing on risky commits (e.g., (Sunghun Kim, Whitehead, and Yi Zhang 2008; Kamei et al. 2013)), we used precision, recall, and F_1 -measure to evaluate our approach. They are computed using TP (true positives), FP (false positives), FN (false negatives), which are defined as follows:

- TP: is the number of defect-commits that were properly classified by BIANCA
- FP: is the number of healthy commits that were classified by BIANCA as risky
- FN: is the number of defect introducing-commits that were not detected by BIANCA
- Precision: $TP / (TP + FP)$
- Recall: $TP / (TP + FN)$
- F_1 -measure: $2 \cdot (\text{precision} \cdot \text{recall}) / (\text{precision} + \text{recall})$

It is worth mentioning that, in the case of defect prevention, false positives can be hard to identify as the defects could be in the code but not yet reported through a bug report (or issue). To address this, we did not include the last six months of history. Following similar studies (Rosen, Grawi, and Shihab 2015; Chen et al. 2014;

Shivaji et al. 2013; Kamei et al. 2013), if a defect is not reported within six months, then it is not considered.

6.4 Empirical Validation

In this section, we show the effectiveness of BIANCA in detecting risky commits using clone detection and project dependency analysis. The main research question addressed by this case study is: *Can we detect risky commits using code comparison within and across related projects, and if so, what would be the accuracy?*

Table 6 shows the results of applying BIANCA in terms of the organization, project name, a short description of the project, the number of classes, the number of commits, the number of defect-commits, the number of defect-commits detected by BIANCA, precision (%), recall (%), F_1 -measure and the average difference, in days, between detected commit and the *original* commit inserting the defect for the first time.

With $\alpha = 35\%$, BIANCA achieves, on average, a precision of 90.75% (13,899/15,316) commits identified as risky. These commits triggered the opening of an issue and had to be fixed later on. On the other hand, BIANCA achieves, on average, 37.15% recall (15,316/41,225), and an average F_1 measure of 52.72%.

Table 6: BIANCA results in terms of organization, project name, a short description, number of class, number of commits, number of defect introducing commits, number of risky commit detected, precision (%), recall (%), F₁-measure (%), the average similarity of first 3 and 5 proposed fixes with the actual fix and the average time difference between detected and original.

Organization	Project Name	Short Description	NoC	#Commits	Bug			Precision	Recall
					Introducing	Detected	Commit		
Alibaba	druid	Database connection pool	3,309	4,775	1,260	787		88.44	62.46
	dubbo	RPC framework	1,715	1,836	119	61		96.72	51.26
	fastjson	JSON parser/generator	2,002	1,749	516	373		95.71	72.29
	jstorm	Stream Process	1,492	215	24	21		90.48	87.50
Apache	hadoop	Distributed processing	9,108	14,154	3,678	851		86.84	23.14
	storm	Realtime system	2,209	7,208	951	444		86.26	46.69
Clojure	clojure	Programming language	335	2,996	596	46		86.96	7.72
Dropwizard	dropwizard	RESTful web services	964	3,809	581	179		96.65	30.81
	metrics	JVM metrics	335	1,948	331	129		95.35	38.97
Eclipse	che	Eclipse IDE	7,818	1,826	169	9		88.89	5.33
Excilys	Android Annotations	Android Development	1,059	2,582	566	9		100.00	1.59
Facebook	fresco	Images Management	1,007	744	100	68		92.65	68.00
GoCD	goCD	Continuous Delivery server	16,735	3,875	499	297		91.58	59.52
Google	auto	source code generators	257	668	124	95		100.00	76.61
	guava	Google Libraries for Java 6+	1,731	3,581	973	592		98.48	60.84
	guice	Dependency injection	716	1,514	605	104		85.58	17.19
	iosched	Android App	1,088	129	9	6		100.00	66.67
Gradle	gradle	Build system	11,876	37,207	6,896	1,557		97.50	22.58
Jankotek	mapdb	Concurrent datastructures	267	1,913	691	440		94.32	63.68
Jhy	jsoup	Parser	136	917	254	153		87.58	60.24

The relatively *low* recall is to be expected since BIANCA classifies commits as risky only if a similar defect-introducing commit happened in one of the 42 open-source projects.

Also, out of the 15,316 commits BIANCA classified as *risky*, only 1,320 (8.6%) were because they were matching a defect-commit inside the same project. This finding supports the idea that developers of a project are not likely to introduce the same defect twice while developers of different projects that share dependencies are, in fact, likely to introduce similar defects. We believe this is an important finding for researchers aiming to achieve cross-project defect prevention, regardless of the technique (e.g., statistical model, AST comparison, code comparison, etc.) employed.

It is important to note that we do not claim that 37.15% of issues in open-source systems are caused by project dependencies. To support such a claim, we would need to analyse the 15,316 detected defect-commits and determine how many yield defects that are similar across projects.

Studying the similarity of defects across projects is a complex task and may require analysing the defect reports manually. This is left as future work. That said, we showed, in this paper, that software systems sharing dependencies also share common issues, irrespective of whether these issues represent similar defects or not.

The experiments took nearly three months using 48 Amazon Virtual Private Servers running in parallel. When deployed, the most time consuming part of BIANCA was spent on building the model of known bug-introducing commits. Once the model was built, it took, on average, 72 seconds to analyze an incoming commit on a typical workstation (quad-core @ 3.2GHz with 8 GB of RAM).

In the following subsections, we compare BIANCA with a random classifier, analyze the best and worst performing projects and assess the quality of the proposed fixes.

6.4.1 Baseline Classifier Comparison

Although our average F_1 measure of 52.72% may seem low at first glance, achieving a high F_1 measure for unbalanced data is very difficult (Menzies et al. 2007). Therefore, a common approach to ground detection results is to compare it to a simple baseline.

To the best of our knowledge, this is the first approach that relies on code similarity instead of code metrics or process metrics for the detection of risky commits.

Comparing it to other approaches will not be accurate. In addition, existing metric-based techniques (e.g., (Nam, Pan, and Kim 2013)) detect risky commits within single projects only. BIANCA, on the other hand, operates across projects. We compared BIANCA with a random classifier to have a baseline and show that we perform better than a simple baseline.

The baseline classifier first generates a random number n between 0 and 1 for the 165,912 commits composing our dataset. For each commit, if n is greater than 0.5, then the commit is classified as risky and vice versa. As expected by a random classifier, our implementation detected $\sim 50\%$ (82,384 commits) of the commits to be *risky*. It is worth mentioning that the random classifier achieved 24.9% precision, 49.96% recall and 33.24% F_1 -measure. Since our data is unbalanced (i.e., there are many more *healthy* than *risky* commits), these numbers are to be expected for a random classifier. Indeed, the recall is very close to 50% since a commit can take on one of two classifications, risky or non-risky. While analysing the precision, however, we can see that the data is unbalanced (a random classifier would achieve a precision of 50% on a balanced dataset).

It is important to note that the purpose of this analysis is not to say that we outperform a simple random classifier, rather to shed light on the fact that our dataset is unbalanced and achieving an average $F_1 = 52.72\%$ is non-trivial, especially when a baseline only achieves an F_1 -measure of 33.24%.

6.4.2 Performance of BIANCA

In this section, we provide insight on the performance of BIANCA by examining the projects for which the best and worst results were obtained.

BIANCA performed best when applied to three projects: Otto by Square (100.00% precision and 76.61% recall, 96.55% F_1 -measure), JStorm by Alibaba (90.48% precision, 87.50% recall, 88.96% F_1 -measure), and Auto by Google (90.48% precision, 87.50% recall, 86.76% F_1 -measure). It performed worst when applied to Android Annotations by Excilys (100.00% precision, 1.59% recall, 3.13% F_1 -measure) and Che by Eclipse (88.89% precision, 5.33% recall, 10.05% F_1 -measure), Openhab by Openhab (100.00% precision, 7.14% recall, 13.33% F_1 -measure). To understand the performance of BIANCA, we conducted a manual analysis of the commits classified as *risky* by BIANCA for these projects.

Otto by Square (F_1 -measure = 96.5%)

At first, the F_1 -measure of Otto by Square seems surprising given the specific set of features it provides. Otto provides a Guava-based event bus. While it does have dependencies that make it vulnerable to defects in related projects, the fact that it provides specific features makes it, at first sight, unlikely to share defects with other projects. Through our manual analysis, we found that out of the 16 *risky* commits detected by BIANCA, 11 (68.75%) matched defect-introducing commits inside the Otto project itself. This is significantly higher than the average number of single-project defects (8.6%). Further investigation of the project management system revealed that very few issues have been submitted for this project (15) and, out of the 11 matches inside the Otto project, 7 were aiming to fix the same issue that had been submitted and fixed several times instead of re-opening the original issue.

JStorm by Alibaba (F_1 -measure = 88.96%)

For JStorm by Alibaba, our manual analysis of the *risky* commits revealed that, in addition to providing stream processes, JStorm mainly supports JSON. The commits detected as *risky* were related to the JSON encoding/decoding functionalities of JStorm. In our dataset, we have several other projects that support JSON encoding and decoding such as FastJSON by Alibaba, Hadoop by Apache, Dropwizard by Dropwizard, Gradle by Gradle and Anthelion by Yahoo. There is, however, only one project supporting JSON in the same cluster as JStorm, Fastjson by Alibaba. FastJSON has a rather large history of defect-commits (516) and 18 out of the 21 commits marked as *risky* by BIANCA were marked so because they matched defect-commits in the FastJSON project.

Auto by Google (F_1 -measure = 86.76%)

Google Auto is a code generation engine. This code generation engine is used by other Google projects in our database, such as Guava and Guice. Most of the Google Auto *risky* commits (79%) matched commits in the Guava and the Guice project. As Guice and Guava share the same code-generation engine (Auto), it makes sense that code introducing defects in these projects share the characteristics of commits introducing defects in Auto.

Openhab by Openhab (F_1 -measure = 13.33%)

Openhab by Openhab provides bus for home automation or smart homes. This is a very specific set of feature. Moreover, Openhab and its dependencies are alone in the green cluster. In other words, the only project against which BIANCA could have checked for matching defects is Openhab itself. BIANCA was able to detect 2/28 bugs for Openhab. We believe that if we had other home-automation projects in our dataset (such as *HomeAutomation* a component based for smart home systems (Seinturier et al. 2012)) then we would have achieved a better F_1 -measure.

Che by Eclipse (F_1 -measure = 10.05%)

Eclipse Che is part of the Eclipse IDE. Eclipse provides development support for a wide range of programming languages such as C, C++, Java and others. Despite the fact that the Che project has a decent amount of defect-commits (169) and that it is in the blue cluster (dominated by Apache) BIANCA was only able to detect nine *risky* commits. After manual analysis of the 169 defect-commits, we were not able to draw any conclusion on why we were not able to achieve better performance. We can only assume that Eclipse’s developers are particularly careful about how they use their dependencies and the quality of their code in general. Only 2% (169/7,818) of their commits introduce new defects.

Annotations by Excilys (F_1 -measure = 3.13%)

The last project we analysed manually is Annotations by Excilys. Similar to Openhab by Openhab, it provides a very particular set of features, which consist of Java annotations for Android projects. We do not have any other project related to Java annotations or the Android ecosystem at large. This caused BIANCA to perform poorly.

Our interpretation of the manual analysis of the best and worst performing projects is that BIANCA performs best when applied to clusters that contain projects that are similar in terms of features, domain or intent. These projects tend to be interconnected through dependencies. In the future, we intend to study the correlation between the cluster betweenness measure and the performance of BIANCA.

6.4.3 Analysis of the Quality of the Fixes Proposed by BIANCA

One of the advantages of BIANCA over other techniques is that it also proposes fixes for the *risky* commits it detects. In order to evaluate the quality of the proposed fixes, we compare the proposed fixes with the actual fixes provided by the developers. To do so, we used the same preprocessing steps we applied to incoming commits: extract, pretty-print, normalize and filter the blocks modified by the proposed and actual fixes. Then, the blocks of the actual fixes and the proposed fixes can be compared with our clone comparison engine.

Similar to other studies recommending fixes, we assess the quality of the first three and five proposed fixes (Pan, Kim, and Whitehead 2008; Kim et al. 2013; Tao et al. 2014; Dallmeier, Zeller, and Meyer 2009; Le Goues et al. 2012; Le, Le, and Lo 2015). The average similarity of the first three fixes is 44.17% while the similarity of the first five fixes is 40.78%. Results are reported in Table 6.

In the framework of this study, for a fix to be ranked as qualitative it has to reach $\alpha=35\%$ similarity threshold. Meaning that the proposed fixed must be at least 35% similar to the actual fix. On average, the proposed fixes are above the $\alpha=35\%$ threshold. On a per-commit basis, BIANCA proposed 101,462 fixes for the 13,899 true positives *risky commits* (7.3 per commit). Out of the 101,462 proposed fixes, 78.67% are above $\alpha=35\%$ threshold.

In other words, BIANCA is able to detect *risky* commits with 90.75% precision, 37.15% recall, and proposes fixes that contain, on average, 40-44% of the actual code needed to transform the *risky* commit into a *non-risky* one.

To further assess the quality of the fixes proposed by BIANCA, we randomly took 250 BIANCA-proposed fixes and manually compared them with the actual fixes provided by the developers. For each fix, we looked at the proposed modifications (i.e., code diff) and the actual modification made by the developer of the system to fix the bug.

We were able to identify the statements from the proposed fixes that can be reused to create fixes similar to the ones that developers had proposed in 84% of the cases. For the remaining cases, it was difficult to understand the changes that the developers made, mainly because of our lack of familiarity with the systems under study. We recognize that a better evaluation of the quality of BIANCA-proposed fixes would be to conduct a user study. We intend to do this as part of future work. In what follows,

```

@@ -293,7 +293,9 @@ private Android(
    byte[] alpnResult = (byte[])
        ↪ getAlpnSelectedProtocol.invoke(socket);
        if (alpnResult != null) return ByteString.of(
    ↪ alpnResult);
    }
-    return ByteString.of((byte[])
    ↪ getNpnSelectedProtocol.invoke(socket));
+    byte[] npnResult = (byte[]) getNpnSelectedProtocol.
    ↪ invoke(socket);
+    if (npnResult == null) return null;
+    return ByteString.of(npnResult);
    } catch (InvocationTargetException e) {
        throw new RuntimeException(e);
    } catch (IllegalAccessException e) {

```

Figure 21: okhttp commit #0ca4c82dd1032625831a5814ea2ddcf165029bdc

we present examples of BIANCA-proposed fixes that were detected as similar to fixes proposed by developers.

In Figures 21 and 22, we show two commits that belong to the Okhttp and Druid systems, respectively. In these figures, the statements shown in red are the ones that triggered the match between the two commits. The Okhttp commit was submitted in February 2014, while the one from Druid was submitted in April 2016. The Druid commit was introduced to fix a bug, which was caused by a prior commit, submitted in March 2016. The bug consisted of invoking a function on a null reference, which led to a null pointer exception, causing the system to crash. This bug could have been avoided if the Druid developers had access to the Okhttp commit.

In a second example, we present a case where BIANCA could have been used to avoid inserting a bug related to race conditions in multi-threaded code.

In Figures 23 and 24, we show two commits that belong to the Netty and Okhttp systems, respectively. For Figure 23, we present an excerpt of the commit that triggered the match. The whole commit affected 44 files with 1,994 additions and 1,335

```

@@ -760,12 +760,14 @@ protected String aliasWrap(String name)
    ↪ {
        Map<String, String> aliasMap = getAliasMap();

        if (aliasMap != null) {
+           if (aliasMap.get(name) == null) {
+               return null;
+           }

-           if (aliasMap.containsKey(name)) {
+           if (aliasMap.containsKey(name)
+               && aliasMap.get(name) != null) {
                return aliasMap.get(name);
            }

            String name_lcase = name.toLowerCase();
-           if (name_lcase != name && aliasMap.containsKey(
    ↪ name_lcase)) {
+           if (name_lcase != name && aliasMap.containsKey(
    ↪ name_lcase)
+               && aliasMap.get(name_lcase) != null)
    ↪ {
                return aliasMap.get(name_lcase);
            }

```

Figure 22: Druid commit #1091861bb15876131653191ae409a523aa8ec0c5

```

+         try {
+             Object v = threadLocalMap.indexedVariable(
+                 ↪ variablesToRemoveIndex);
+             if (v != null && v != InternalThreadLocalMap.
+                 ↪ UNSET) {
+                 @SuppressWarnings("unchecked")
+                 Set<FastThreadLocal<?>> variablesToRemove =
+                 ↪ (Set<FastThreadLocal<?>>) v;
+                 FastThreadLocal<?>[] variablesToRemoveArray
+                 ↪ =
+                 variablesToRemove.toArray(new
+                 ↪ FastThreadLocal[variablesToRemove.size()]);
+                 for (FastThreadLocal<?> tlv :
+                 ↪ variablesToRemoveArray) {
+                     tlv.remove(threadLocalMap);
+                 }
+             }
+         } catch (IOException e) {
+         } catch (InterruptedException e) {
+         } finally {
+             InternalThreadLocalMap.remove();
+         }

```

Figure 23: netty commit #085a61a310187052e32b4a0e7ae9700dbe926848

```

@@ -682,16 +682,21 @@ private void handleWebSocketUpgrade(
    ↪ Socket socket ,
    BufferedSource source , Buffer
    response.getWebSocketListener().onOpen(webSocket ,
    ↪ fancyResponse);
    String name = "MockWebServer WebSocket " + request.
    ↪ getPath();
    webSocket.initReaderAndWriter(name, 0, streams);
-    webSocket.loopReader();
-
-    // Even if messages are no longer being read we need to
    ↪ wait
    for the connection close signal.
    try {
-        connectionClose.await();
-    } catch (InterruptedException ignored) {
-    }
+        webSocket.loopReader();

-    closeQuietly(sink);
-    closeQuietly(source);
+    // Even if messages are no longer being read we need
    ↪ to wait
    for the connection close signal.
+    try {
+        connectionClose.await();
+    } catch (InterruptedException ignored) {
+    }
+
+    } catch (IOException e) {
+        webSocket.failWebSocket(e, null);
+    } finally {
+        closeQuietly(sink);
+        closeQuietly(source);
+    }
    }

```


deletions. The Netty commit was submitted in June 2014 while the one from OKHttp was submitted in January 2017. The bug consisted of resource leakage in a multi-threaded environment. The similarity between the two commits comes from the `try` and `catch` blocks associated with the used exceptions, more precisely, the fact of freeing resources in case a thread crashes with a `finally` block to follow the `try` and `catch` blocks. In the `try` block, the threads are launched and, in case an exception happens, the `catch` block is executed. However, if the developer closes the resources consumed by the thread at the end of the `try` block then, in the case of an exception, the resources would not be freed. Instead of duplicating the resource management code in the `try` and `catch` blocks, a good practice would be to have it in a `finally` block that always executes itself, regardless of whether an exception is thrown or not. In the commit presented by Figure 23, we can see that a large refactoring has been done in order to prevent crashed threads to keep using resources. This bug could have been avoided if the Okhttp developers had access to the Netty commit.

Another example is the one depicted in Figures 25 and 26, showing two commits that belong to the JSoup and Orientdb systems, respectively. The first commit was submitted in November 2013, while the Orientdb was submitted two years later in October 2015. The Orientdb commit was used to fix a bug introduced by a commit that was submitted earlier in October 2015. This bug would have been avoided if the developer had access to the JSoup commit, that is proposed by BIANCA as the closest match.

In these fixes, we can see that the developers are working with the `StringBuilder` class. According to the Java documentation, the `StringBuilder` class provides an API compatible with `StringBuffer`, but with no guarantee of synchronization. This class is designed for use as a drop-in replacement for `StringBuffer` in places where the string buffer was being used by a single thread. Where possible, it is recommended that this class be used in preference to `StringBuffer` as it will be faster under most implementations. Developers usually use the `StringBuilder` class to build strings using the `append` and `insert` methods. Using the `StringBuilder` class rather than plain string concatenation (i.e., using the `+` operator) is known to be a good Java practice as it improves performance.

In both cases, the code has been modified to avoid the appending of `null` string. In JSoup, it is done by the method `shouldCollapseAttribute`, which checks for

```

@@ -34,7 +35,11 @@ public Object jjtAccept(OrientSqlVisitor
    ↪ visitor, Object data) {
        public void toString(Map<Object, Object> params,
            ↪ StringBuilder builder) {
            expression.toString(params, builder);
            builder.append(" _MATCHES_");
-        builder.append(right);
+        if(right!=null) {
+            builder.append(right);
+        } else {
+            rightParam.toString(params, builder);
+        }
    }
}

```

Figure 25: OrientDB commit #444db817ee9404b17c1208df51781ce9cb6a2666

empty values. In Orientdb, the same operation is performed by a simple null check on the string named `right`. Note that this kind of *bug* would not have been spotted by a static analysis tool such as PMD (Dangel 2000) because it is *legal* to pass a null string as a parameter of function expecting a string. In both cases, however, the developers were tasked to avoid the appending of null strings.

6.5 Threats to Validity

In this section, we propose a discussion on limitations and threats to validity.

We identified three main limitations of our approach, BIANCA, that require further studies.

BIANCA is designed to work on multiple related systems. Applying BIANCA on a single system will most likely be ineffective; it is unlikely to have a large number of similar bugs within the same system. For single systems, we recommend the use of statistical models based on process and code metrics for the detection of risky commits such as the ones developed by Kamei et al. and Rosen et al. (Kamei et al. 2013; Rosen, Grawi, and Shihab 2015). A metric-based solution, however, may turn to be ineffective when applied across systems because of the difficulty associated with

```

@@ -100,6 +111,15 @@ protected void html(StringBuilder
    ↪ accum, Document.OutputSettings out) {
-        accum
-            .append(key)
-            .append("=\")
-            .append(Entities.escape(value, out))
-            .append("\");
+        accum.append(key);
+        if (!shouldCollapseAttribute(out)) {
+            accum.append("=\");
+            Entities.escape(accum, value, out, true, false
    ↪ , false);
+        accum.append(' ');
+    }
}

/**
protected boolean isDataAttribute() {
    return key.startsWith(Attributes.dataPrefix) && key
    ↪ .length()
    > Attributes.dataPrefix.length();
}

+ /**
+  * Collapsible if it's a boolean attribute and value is
    ↪ empty or same as name
+  */
+ protected final boolean shouldCollapseAttribute(
    ↪ Document.OutputSettings out) {
+     return ("".equals(value) || value.equalsIgnoreCase(
    ↪ key))
+         && out.syntax() || Document.OutputSettings.
    ↪ Syntax.html
+         && Arrays.binarySearch(booleanAttributes,
    ↪ key) >= 0;
+ }
+

```

identifying common thresholds that are applicable to a wide range of systems.

The second limitation is related to scalability of the approach. Because BIANCA operates on multiple systems, we need to build a model that comprises all their commits, which is a time consuming process. It took nearly three months using 48 Amazon Virtual Private Servers running in parallel to build and test the model for our experiments.

The third limitation we identified has to do with the fact that BIANCA is designed to work with Java systems only. It is however common to have a multitude of programming languages used in an environment with many inter-related systems. We intend to extend BIANCA to process commits from other languages as well.

The selection of target systems is one of the common threats to validity for approaches aiming to improve the analysis of software systems. It is possible that the selected programs share common properties that we are not aware of and therefore, invalidate our results. However, the systems analyzed by BIANCA were selected from Github based on their popularity and the ability to mine their past issues and to retrieve their dependencies. Any project that satisfies these criteria would be included in the analysis. Moreover, the systems vary in terms of purpose, size, and history.

In addition, we see a threat to validity that stems from the fact that we only used open-source systems. The results may not be generalizable to industrial systems. We intend to undertake these studies in future work.

The programs we used in this study are all based on the Java programming language. This can limit the generalization of the results to projects written in other languages. However, similar to Java, one can write a TXL grammar for a new language then BIANCA can work since BIANCA relies on TXL.

Moreover, we use NICAD as the code comparison engine. The accuracy of NICAD affects the accuracy of BIANCA. This said, since NICAD has been tested on large systems, we are confident that it is a suitable engine for comparing code using TXL. Also, there is nothing that prevents from using other text-based code comparisons engines. Another threat related to the use of NICAD is the use of 35% as a similarity threshold. A different threshold may affect the results. We chose this threshold because it resulted in the best trade-off between precision and recall when analysing a subset of the dataset.

Finally, part of the analysis of the BIANCA proposed fixes that we did was based

on manual comparison of the BIANCA fixes with those proposed by developers. Although we exercised great care in analyzing all the fixes, we may have misunderstood some aspects of the commits.

In conclusion, internal and external validity have both been minimized by choosing a set of 42 different systems, using input data that can be found in any programming languages and version systems (commit and changesets).

6.6 Chapter Summary

In this chapter, we presented BIANCA (Bug Insertion ANTicipation by Clone Analysis at commit time), an approach that detects risky commits (i.e., a commit that is likely to introduce a bug) with an average of 90.75% precision and 37.15% recall. BIANCA uses clone detection techniques and project dependency analysis to detect risky commits within and across projects. BIANCA operates at commit-time, i.e., before the commits reach the central code repository. In addition, because it relies on code comparison, BIANCA does not only detect risky commits but also makes recommendations to developers on how to fix them. We believe that this makes BIANCA a practical approach for preventing bugs and proposing corrective measures that integrate well with developers workflow through the commit mechanism.

In the next chapter, we present CLEVER, an approach that combines metric and clone based classifications in order to address the scalability issues of BIANCA.

Chapter 7

Combining Code Metrics with Clone Detection for Just-In-Time Fault Prevention and Resolution

7.1 Introduction

In the previous chapter, we presented BIANCA, an approach that relies on clone-detection to detect risky commits and propose fixes. While the performances of BIANCA are satisfactory, it still has major limitations. For instance, it only supports the Java programming language (though it could be expanded to support other languages) and building the model supporting it is incredibly expensive.

There exist techniques that aim to detect risky commits (e.g., (Briand, Daly, and Wust 1999; Chidamber and Kemerer 1994; Subramanyam and Krishnan 2003)), among which the most recent approach is the one proposed by Rosen et al. (Rosen, Grawi, and Shihab 2015). The authors developed an approach and a supporting tool, Commit-guru, that relies on building models from historical commits using code and process metrics (e.g., code complexity, the experience of the developers, etc.) as main features. These models are used to classify new commits as risky or not. Commit-guru has been shown to outperform previous techniques (e.g., (Kamei et al. 2013; Kpodjedo et al. 2010)).

However, Commit-guru and similar tools suffer from some limitations. First, they tend to generate high false positive rates by classifying healthy commits as risky. The

second limitation is that they do not provide recommendations to developers on how to fix the detected risky commits. They simply return measurements that are often difficult to interpret by developers. In addition, they tend to be slow, despite being based on code metrics rather than code comparison because they are not incremental. Indeed, they require pulling the complete history of a project for each analysis. On projects with tens of thousands of commit, this is problematic. Moreover, this tools only support git when the source-code versioning landscape is diverse. As shown by the 2018 StackOverflow survey, companies use Subversion, Team Foundation, Mercurial or other tools at 16.6%, 11.3%, 3.7% and, 19.1% respectively. Our industrial partner for this project uses Git and Mercurial, so we required a more versatile approach. Another shortcomming of Commit-Guru is that it only uses the commit-message to determinate if a commit is a fix rather than the issue control system. This is problematic because developers leverage shortcut hardcoded in commit-message to close issues automatically. For example, the following commit-message `fix #7` would be interpreted by Commit-Guru as a bug-fixing commit regardless of the fact issue #7, on the issue control system could be categorized as a feature to implement. In this case, the word `fix` was merely used to trigger the automatic closing of the issue.

Finally, they have been mainly validated using open source systems. Their effectiveness, when applied to industrial systems, has yet to be shown.

In this chapter, we propose an approach, called CLEVER (Combining Levels of Bug Prevention and Resolution techniques), that relies on a two-phase process for intercepting risky commits before they reach the central repository. The first phase consists of building a metric-based model to assess the likelihood that an incoming commit is risky or not. This is similar to existing approaches. The next phase relies on clone detection to compare code blocks extracted from suspicious risky commits, detected in the first phase, with those of known historical fault-introducing commits. This additional phase provides CLEVER with two apparent advantages over Commit-guru. First, as we will show in the evaluation section, CLEVER is able to reduce the number of false positives by relying on code matching instead of mere metrics. The second advantage is that, with CLEVER, it is possible to use commits that were used to fix faults introduced by previous commits to suggest recommendations to developers on how to improve the risky commits at hand. This way, CLEVER goes one step further than Commit-guru (and similar techniques) by providing developers

with a potential fix for their risky commits.

Another important aspect of CLEVER is its ability to detect risky commits not only by comparing them to commits of a single project but also to those belonging to other projects that share common dependencies. This is important in the context of an industrial setting where software systems tend to have many dependencies that make them vulnerable to the same faults.

CLEVER was developed in collaboration with software developers from Ubisoft La Forge. Ubisoft is one of the world’s largest video game development companies specializing in the design and implementation of high-budget video games. Ubisoft software systems are highly coupled containing millions of files and commits, developed and maintained by more than 8,000 developers scattered across 29 locations in six continents.

We tested CLEVER on 12 major Ubisoft systems. The results show that CLEVER can detect risky commits with 79% precision and 65% recall, which outperforms the performance of Commit-guru (66% precision and 63% recall) when applied to the same dataset. In addition, 66.7% of the proposed fixes were accepted by at least one Ubisoft software developer, making CLEVER an effective and practical approach for the detection and resolution of risky commits.

7.2 Approach

Figures 27, 28 and 29 show an overview of the CLEVER approach, which consists of two parallel processes.

In the first process (Figures 27 and 28), CLEVER manages events happening on project tracking systems to extract fault-introducing commits and commits and their corresponding fixes. For simplicity reasons, in the rest of this paper, we refer to commits that are used to fix defects as *fix-commits*. We use the term *defect-commit* to mean a commit that introduces a fault.

The project tracking component of CLEVER listens to bug (or issue) closing events of Ubisoft projects. Currently, CLEVER is tested on 12 large Ubisoft projects. These projects share many dependencies. We clustered them based on their dependencies with the aim to improve the accuracy of CLEVER. This clustering step is

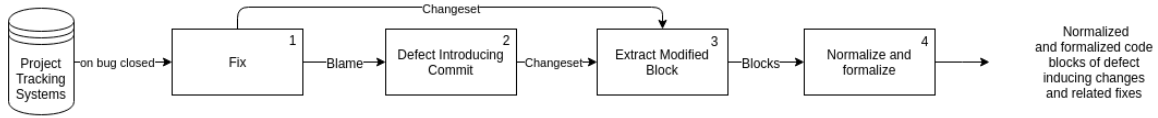


Figure 27: Managing events happening on project tracking systems to extract defect-introducing commits and commits that provided the fixes

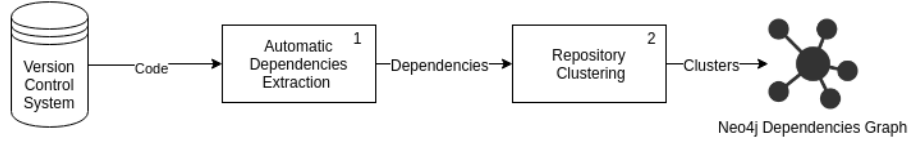


Figure 28: Clustering by dependency

important in order to identify faults that may exist due to dependencies, while enhancing the quality of the proposed fixes.

In the second process (Figure 29), CLEVER intercepts incoming commits before they leave a developer’s workstation using the concept of pre-commit hooks.

Ubisoft’s developers already use pre-commit hooks for all sorts of reasons such as identifying the tasks that are addressed by the commit at hand, specifying the reviewers who will review the commit, and so on. Implementing this part of CLEVER as a pre-commit hook is an important step towards the integration of CLEVER with the workflow of developers at Ubisoft. The developers do not have to download, install, and understand additional tools in order to use CLEVER.

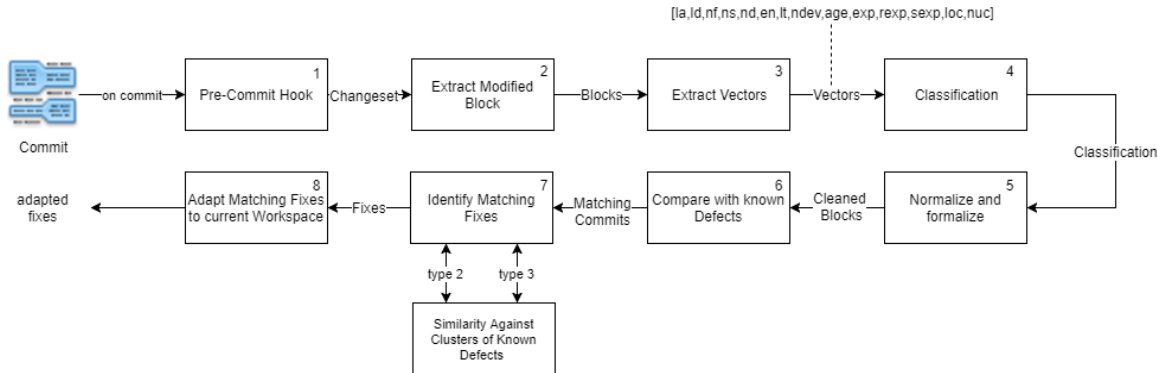


Figure 29: Classifying incoming commits and proposing fixes

Once the commit is intercepted, we compute code and process metrics associated with this commit. The selected metrics are discussed further in Section 7.2.2. The result is a feature vector (Step 4) that is used for classifying the commit as *risky* or *non-risky*.

If the commit is classified as *non-risky*, then the process stops, and the commit can be transferred from the developer’s workstation to the central repository. *Risky* commits, on the other hand, are further analysed in order to reduce the number of false positives (healthy commits that are detected as risky). We achieve this by first extracting the code blocks that are modified by the developer and then compare them to code blocks of known fault-introducing commits.

7.2.1 Clustering Projects

We cluster projects using the same technic as BIANCA (i.e. Newman algorithm (Girvan and Newman 2002; Newman and Girvan 2004) on the dependencies) for the same reasons (i.e. projects that share dependencies are most likely to contain defects caused by misuse of these dependencies).

Within the context of Ubisoft, dependencies can be *external* or *internal* depending on whether the products are created in-house or supplied by a third-party. For confidentiality reasons, we cannot reveal the name of the projects involved in the project dependency graph. We show the 12 projects in yellow color with their dependencies in blue color in Figure 30. In total, we discovered 405 distinct dependencies. Dependencies can be internal (i.e. library developed at Ubisoft) or external (i.e. library provided by third parties). The resulting partitioning is shown in Figure 31.

At Ubisoft, dependencies are managed within the framework of a single repository, which makes their automatic extraction possible. The dependencies could also be automatically retrieved if the projects use a dependency manager such as Maven.

7.2.2 Building a Database of Code Blocks of Defect-Commits and Fix-Commits

In order to build of database of code blocks of defect-commits and fix-commits we use the same technic as for BIANCA. First, we listen to issue closing-events and extract the blocks of code belonging to incrimated commits (i.e. commit known to

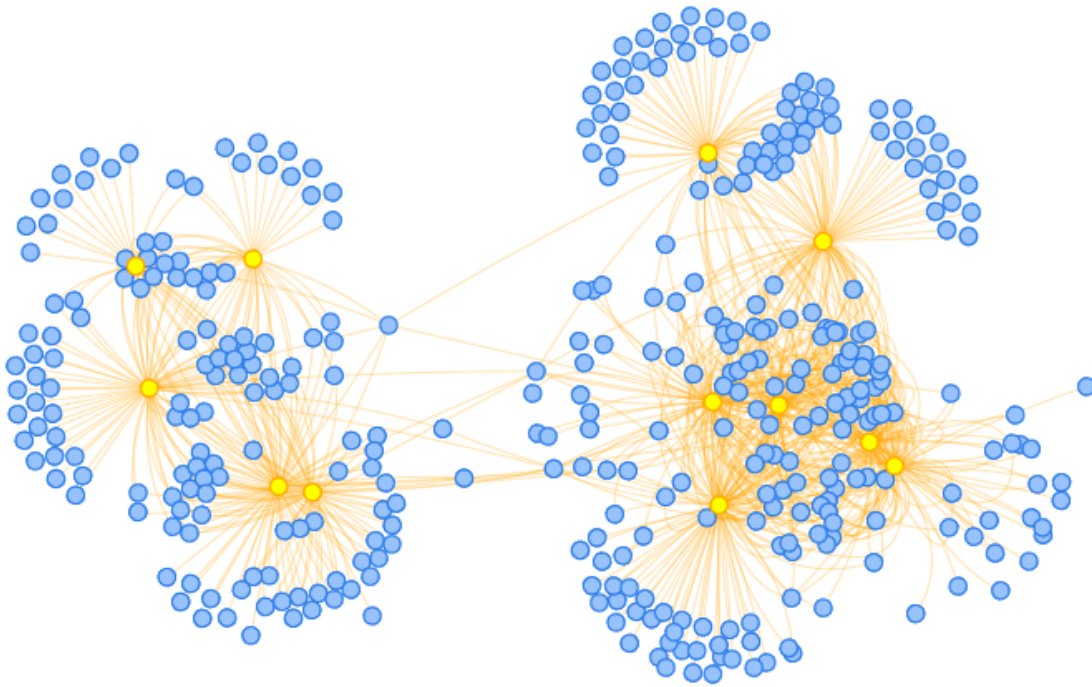


Figure 30: Dependency Graph

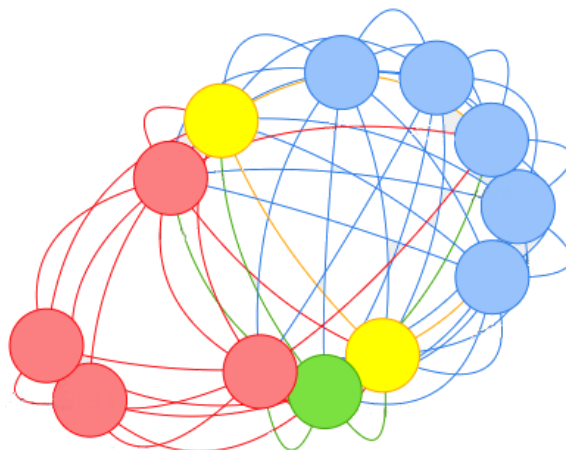


Figure 31: Clusters

have introduced a bug) using our refined version of TXL and NICAD.

7.2.3 Building a Metric-Based Model

We adapted Commit-guru (Rosen, Grawi, and Shihab 2015) for building the metric-based model. Commit-guru uses a list of keywords proposed by Hindle *et al.* (Hindle, German, and Holt 2008) to classify commit in terms of *maintenance*, *feature* or *fix*. Then, it uses the SZZ algorithm to find the defect-commit linked to the fix-commit. For each defect-commit, Commit-guru computes the following code metrics: *la* (lines added), *ld* (lines deleted), *nf* (number of modified files), *ns* (number of modified subsystems), *nd* (number of modified directories), *en* (distribution of modified code across each file), *lt* (lines of code in each file (sum) before the commit), *ndev* (the number of developers that modified the files in a commit), *age* (the average time interval between the last and current change), *exp* (number of changes previously made by the author), *rexp* (experience weighted by age of files ($1 / (n + 1)$)), *sexp* (previous changes made by the author in the same subsystem), *loc* (total number of modified LOC across all files), *nuc* (number of unique changes to the files). Then, a statistical model is built using the metric values of the defect-commits. Using linear regression, Commit-guru is able to predict whether incoming commits are *risky* or not.

We had to modify Commit-guru to fit the context of this study. First, we used information found in Ubisoft’s internal project tracking system to classify the purpose of a commit (i.e., *maintenance*, *feature* or *fix*). In other words, CLEVER only classifies a commit as a defect-commit if it is the root cause of a fix linked to a crash or a bug in the internal project tracking system. Using internal pre-commit hooks, Ubisoft developers must link every commit to a given task #ID. If the task #ID entered by the developer matches a bug or crash report within the project tracking system, then we perform the SCM blame/annotate function on all the modified lines of code for their corresponding files on the fix-commit’s parents. This returns the commits that previously modified these lines of code and are flagged as defect-commits. Another modification consists of the actual classification algorithm. We did not use linear regression but instead the random forest algorithm (Tin Kam Ho 1995; Tin Kam Ho 1998). The random forest algorithm turned out to be more effective as described in Section 7.4. Finally, we had to rewrite Commit-guru in GoLang for performance and

internal reasons.

7.2.4 Comparing Code Blocks

Each time a developer makes a commit, CLEVER intercepts it using a pre-commit hook and classifies it as *risky* or not. If the commit is classified as *risky* by the metric-based classifier, then, we extract the corresponding code block (in a similar way as in the previous phase), and compare it to the code blocks of historical defect-commits. If there is a match, then the new commit is deemed to be risky. A threshold α is used to assess the extent beyond which two commits are considered similar.

Once again, we reuse the comparing method approach created for BIANCA when it comes to compare blocks of code.

7.2.5 Classifying Incoming Commits

As discussed in Section 7.2.3, a new commit goes through the metric-based model first (Steps 1 to 4). If the commit is classified as *non-risky*, we simply let it through, and we stop the process. If the commit is classified as *risky*, however, we continue the process with Steps 5 to 8 in our approach.

One may wonder why we needed to have a metric-based model in the first place. We could have resorted to clone detection as the main mechanism as exposed in the previous chapter. The main reason for having the metric-based model is efficiency. If each commit had to be analysed against all known signatures using code clone similarity, then, it would have made CLEVER impractical at Ubisoft’s scale. We estimate that, in an average workday (i.e. thousands of commits), if all commits had to be compared against all signatures on the same cluster we used for our experiments it would take around 25 minutes to process a commit with the current processing power dedicated to *CLEVER*. In comparison, it takes, on average, 3.75 seconds with the current two-step approach.

7.2.6 Proposing Fixes

An important aspect in the design of CLEVER is the ability to provide guidance to developers on how to improve risky commits. We achieve this by extracting from the database the fix-commit corresponding to the top 1 matching defect-commits

and present it to the developer. We believe that this makes CLEVER a practical approach. Developers can understand why a given modification has been reported as risky by looking at code instead of simple metrics as in the case of the studies reported in (Kamei et al. 2013; Rosen, Grawi, and Shihab 2015).

Finally, using the fixes of past defects, we can provide a solution, in the form of a contextualised diff, to developers. A contextualised diff is a diff that is modified to match the current workspace of the developer regarding variable types and names. In Step 8 of Figure 3, we adapt the matching fixes to the actual context of the developer by modifying indentation depth and variable name in an effort to reduce context switching. We believe that this would make it easier for developers to understand the proposed fixes and see if it applies in their situation.

All the proposed fixes will come from projects in the same cluster as the project where the *risky* commit is. Thus, developers have access to fixes that should be easier to understand as they come from projects similar to theirs inside the company.

7.3 Experimental Setup

In this section, we present the setup of our case study in terms of repository selection, dependency analysis, comparison process and evaluation measures.

7.3.1 Project Repository Selection

In collaboration with Ubisoft developers, we selected 12 major software projects (i.e., systems) developed at Ubisoft to evaluate the effectiveness of CLEVER. These systems continue to be actively maintained by thousands of developers. Ubisoft projects are organized by game engines. A game engine can be used in the development of many high-budget games. The projects selected for this case study are related to the same game engine. For confidentiality and security reasons, neither the names nor the characteristics of these projects are provided. We can, however, disclose that the size of these systems altogether consists of millions of files of code, hundreds of millions of lines of code and hundreds of thousands of commits. All 12 systems are AAA videos games.

7.3.2 Project Dependency Analysis

Figure 30 shows the project dependency graph. As shown in Figure 30, these projects are highly interconnected. A review of each cluster shows that this partitioning divides projects in terms of their high-level functionalities. For example, one cluster is related to a given family of video games, whereas the other cluster refers to another family. We showed this partitioning to 11 experienced software developers and ask them to validate it. They all agreed that the results of this automatic clustering are accurate and reflects well the various project groups of the company. The clusters are used for decreasing the rate of false positive. In addition, fixes mined across projects but within the cluster are qualitative as shown in our experiments.

7.3.3 Building a Database of Defect-Commits and Fix-Commits

To build the database that we can use to assess the performance of CLEVER, we use the same process as discussed in Section 7.2.2. We retrieve the full history of each project and label commits as defect-commits if they appear to be linked to a closed issue using the SZZ algorithm (Kim, Zimmermann, Pan, et al. 2006b). This baseline is used to compute the precision and recall of CLEVER. Each time CLEVER classifies a commit as *risky*; we can check if the *risky* commit is in the database of defect-introducing commits. The same evaluation process is used by related studies (El Emam, Melo, and Machado 2001; Lee et al. 2011; Bhattacharya and Neamtiu 2011; Kpodjedo et al. 2010; Kamei et al. 2013).

7.3.4 Process of Comparing New Commits

Similarly to PRECINCT and CLEVER, the repository is cloned, rewinded and re-played commit by commit in order to evaluate the performances of CLEVER.

While figure 32 explains the processes of *CLEVER* it does not encompass all the cases. Indeed, the user that experiences the defect can be internal (i.e. another developer, a tester, ...) or external (i.e. a player). In addition, many other projects receive commits in parallel and they are all to be compared with all the known signatures.

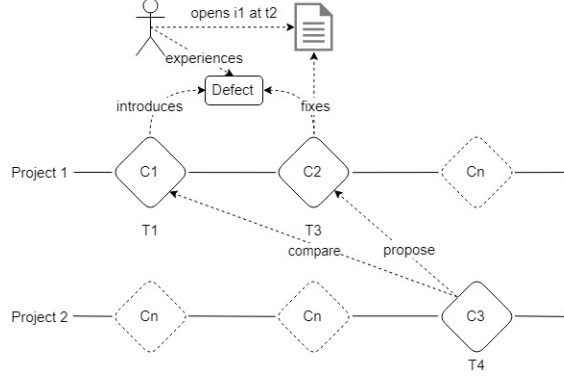


Figure 32: Process of Comparing New Commits

7.4 Empirical Validation

In this section, we show the effectiveness of CLEVER in detecting risky commits using a combination of metric-based models and clone detection. The main research question addressed by this case study is: *Can we detect risky commits by combining metrics and code comparison within and across related Ubisoft projects, and if so, what would be the accuracy?*

The experiments took nearly two months using a cluster of six 12 3.6 Ghz cores with 32GB of RAM each. The most time consuming part of the experiment consists of building the baseline as each commit must be analysed with the SZZ algorithm. Once the baseline was established, the model built, it took, on average, 3.75 seconds to analyse an incoming commit on our cluster.

In the following subsections, we provide insights on the performance of CLEVER by comparing it to Commit-guru (Rosen, Grawi, and Shihab 2015) alone, i.e., an approach that relies only on metric-based models. We chose Commit-guru because it has been shown to outperform other techniques (e.g., (Kamei et al. 2013; Kpodjedo et al. 2010)). Commit-guru is also open source and easy to use.

7.4.1 Performance of CLEVER

When applied to 12 Ubisoft projects, CLEVER detects risky commits with an average precision, recall, and F1-measure of 79.10%, a 65.61%, and 71.72% respectively. For clone detection, we used a threshold of 30%. This is because Roy *et al.* (Roy and

Table 7: Workshop results

	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12
P1	Accepted	Rejected	Accepted	Accepted	Unsure	Accepted	Unsure	Rejected	Rejected	Accepted	Accepted	Unsure
P2	Accepted	Rejected	Accepted	Unsure	Unsure	Accepted	Unsure	Rejected	Rejected	Accepted	Accepted	Unsure
P3	Accepted	Rejected	Accepted	Unsure	Unsure	Accepted	Unsure	Rejected	Rejected	Accepted	Accepted	Unsure
P4	Accepted	Rejected	Accepted	Unsure	Unsure	Accepted	Unsure	Accepted	Rejected	Accepted	Accepted	Unsure
P5	Accepted	Rejected	Accepted	Accepted	Unsure	Accepted	Unsure	Rejected	Rejected	Accepted	Accepted	Unsure
P6	Accepted	Rejected	Accepted	Unsure	Unsure	Accepted	Unsure	Accepted	Rejected	Accepted	Accepted	Unsure

Cordy 2008) showed through empirical studies that using NICAD with a threshold of around 30%, the default setting, provides good results for the detection of Type 3 clones. When applied to the same projects, Commit-guru achieves an average precision, recall, and F1-measure of 66.71%, 63.01% and 64.80%, respectively.

We can see that with the second phase of CLEVER (clone detection) there is considerable reduction in the number of false positives (precision of 79.10% for CLEVER compared to 66.71% for Commit-guru) while achieving similar recall (65.61% for CLEVER compared to 63.01% for Commit-guru).

7.4.2 Analysis of the Quality of the Fixes Proposed by CLEVER

In order to validate the quality of the fixes proposed by CLEVER, we conducted an internal workshop where we invited a number of people from Ubisoft development team. The workshop was attended by six participants: two software architects, two developers, one technical lead, and one IT project manager. The participants have many years of experience at Ubisoft.

The participants were asked to review 12 randomly selected fixes that were proposed by CLEVER. These fixes are related to one system in which the participants have excellent knowledge. We presented them with the original buggy commits, the original fixes for these commits, and the fixes that were automatically extracted by CLEVER. We asked them the following question “*Is the proposed fix applicable in the given situation?*” for each fix.

The review session took around 50 minutes. This does not include the time it took to explain the objective of the session, the setup, the collection of their feedback, etc.

We asked the participants to rank each fix proposed by CLEVER using this scheme:

- Fix Accepted: The participant found the fix proposed by CLEVER applicable

to the risky commit.

- **Unsure:** In this situation, the participant is unsure about the relevance of the fix. There might be a need for more information to arrive to a verdict.
- **Fix Rejected:** The participant found the fix is not applicable to the risky commit.

Table 7 shows answers of the participants. The columns refer to the fixes proposed by CLEVER, whereas the rows refer to the participants that we denote using P1, P2, ..., P6. As we can see from the table, 41.6% of the proposed fixes (F1, F3, F6, F10 and F12) have been accepted by all participants, while 25% have been accepted by at least one member (F4, F8, F11). We analysed the fixes that were rejected by some or all participants to understand the reasons.

F2 was rejected by our participants because the region of the commit that triggered a match is a generated code. Although this generated code was pushed into the repositories as part of bug fixing commit, the root cause of the bug lies in the code generator itself. Our proposed fix suggests to update the generated code. Because the proposed fix did not apply directly to the bug and the question we ask our reviewers was *“Is the proposed fix applicable in the given situation?”* they rejected it. In this occurrence, the proposed fix was not applicable.

F4 was accepted by two reviewers and marked as unsure by the other participants. We believe that this was due the lack of context surrounding the proposed fix. The participants were unable to determine if the fix was applicable or not without knowing what the original intent of the buggy commit was. In our review session, we only provided the reviewers with the regions of the commits that matched existing commits and not the full commit. Full commits can be quite lengthy as they can contain asset descriptions and generated code, in addition to the actual code. In this occurrence, the full context of the commit might have helped our reviewers to decide if *F4* was applicable or not. *F5* and *F7* were classified as unsure by all our participants for the same reasons.

F8 was rejected by four of participants and accepted by two. The participants argued that the proposed fix was more a refactoring opportunity than an actual fix.

F12 was marked as unsure by all the reviewers because the code had to do with a subsystem that is maintained by another team and the participants felt that it was out of the scope of this session.

After the session, we asked the participants two additional questions: *Will you use CLEVER in the future?* and *What aspects of CLEVER need to be improved?*

All the participants answered the first question favourably. They also proposed to embed CLEVER with Ubisoft’s quality assurance tool suite. The participants reported that the most useful aspects of CLEVER are:

- Ability to leverage many years of historical data of inter-related projects, hence allowing development teams to share their experiences in fixing bugs.
- Easy integration of CLEVER into developers’ work flow based on the tool’s ability to operate at commit-time.
- Precision and recall of the tool (79% and 65% respectively) demonstrating CLEVER’s capabilities to catch many defects that would otherwise end up in the code repository. For the second question, the participants proposed to add a feedback loop to CLEVER where the input of software developers is taken into account during classification. The objective is to reduce the number of false negatives (risky commits that are flagged as non-risky) and false positives (non-risky commits that are flagged as risky). The feedback loop mechanism would work as follows: When a commit is misclassified by the tool, the software developer can choose to ignore CLEVER’s recommendation and report the misclassified commit. If the fix proposition is not used, then, we would give that particular pattern less strength over other patterns automatically.

We do not need manual input from the user because CLEVER knows the state of the commit before the recommendation and after. If both versions are identical then we can mark the recommendation as not helpful. This way, we can also compensate for human error (i.e., a developer rejecting CLEVER recommendation when the commit was indeed introducing a defect. We would know this by using the same processes that allowed us to build our database of defect-commits as described in Section 7.2.2. This feature is currently under development.

It is worth noting that Ubisoft developers who participated to this study did not think that CLEVER fixes that were deemed irrelevant were a barrier to the deployment of CLEVER. In their point of view, the performance of CLEVER in terms of classification should make a significant impact as suspicious commits will receive extended reviews and/or further investigations.

We are also investigating the use of adaptive learning techniques to improve the classification mechanism of CLEVER. In addition to this, the participants discussed the limitation of CLEVER as to its inability to deal with automatically generated code. We are currently working with Ubisoft’s developers to address this limitation.

7.5 Threats to Validity

We identified two main limitations of our approach, CLEVER, which require further studies.

CLEVER is designed to work on multiple related systems. Applying CLEVER to a single system will most likely be less effective. The two-phase classification process of CLEVER would be hindered by the fact that it is unlikely to have a large number of similar bugs within the same system. For single systems, we recommend the use of metric-based models. A metric-based solution, however, may turn to be ineffective when applied across systems because of the difficulty associated with identifying common thresholds that are applicable to a wide range of systems.

The second limitation we identified has to do with the fact that CLEVER is designed to work with Ubisoft systems. Ubisoft uses C#, C, C++, Java and other internally developed languages. It is however common to have other languages used in an environment with many inter-related systems. We intend to extend CLEVER to process commits from other languages as well.

The selection of target systems is one of the common threats to validity for approaches aiming to improve the analysis of software systems. It is possible that the selected programs share common properties that we are not aware of and therefore, invalidate our results. Because of the industrial nature of this study, we had to work with the systems developed by the company.

The programs we used in this study are all based on the C#, C, C++ and Java programming languages. This can limit the generalization of the results to projects written in other languages, especially that the main component of CLEVER is based on code clone matching.

Finally, part of the analysis of the CLEVER proposed fixes that we did was based on manual comparisons of the CLEVER fixes with those proposed by developers with a focus group composed of experienced engineers and software architects. Although,

we exercised great care in analysing all the fixes, we may have misunderstood some aspects of the commits.

In conclusion, internal and external validity have both been minimized by choosing a set of 12 different systems, using input data that can be found in any programming languages and version systems (commits and changesets).

7.6 Chapter Summary

In this chapter, we presented CLEVER (Combining Levels of Bug Prevention and Resolution Techniques), an approach that detects risky commits (i.e., a commit that is likely to introduce a bug) with an average of 79.10% precision and a 65.61% recall. CLEVER combines code metrics, clone detection techniques, and project dependency analysis to detect risky commits within and across projects. CLEVER operates at commit-time, i.e., before the commits reach the central code repository. Also, because it relies on code comparison, CLEVER does not only detect risky commits but also makes recommendations to developers on how to fix them. We believe that this makes CLEVER a practical approach for preventing bugs and proposing corrective measures that integrate well with the developer’s workflow through the commit mechanism. CLEVER is still in its infancy and we expect it to be available this year to thousands of developers.

In the next chapter, we present JCHARMING, an approach to reproduce bugs using stacktraces contained in bug report. JCHARMING is meant to be used by developers when the commit-time approaches presented in Chapters 4, 6 and 7 did not manage to prevent the introduction of a defect.

Chapter 8

Bug Reproduction Using Crash Traces and Directed Model Checking

In previous chapters, we presented three approaches (PRECINCT, BIANCA and, CLEVER) that aim to improve the code quality by performing maintenance operations at commit-time. While these approaches have shown good performances, they never will be 100% accurate in preventing defects from reaching the customers and field crash will happen.

The first (and perhaps main) step in understanding the cause of a field crash is to reproduce the bug that caused the system to fail. A survey conducted with developers of major open source software systems such as Apache, Mozilla and Eclipse revealed that one of the most valuable piece of information that can help locate and fix the cause of a crash is the one that can help reproduce it (Bettenburg et al. 2008).

Bug reproduction is, however, a challenging task because of the limited amount of information provided by the end users. There exist several bug reproduction techniques. They can be grouped into two categories: (a) On-field record and in-house replay (Narayanasamy, Pokam, and Calder 2005; Artzi, Kim, and Ernst 2008; Jaygarl et al. 2010), and (b) In-house crash explanation [Manevich, Sridharan, and Adams (2004); chandra2009snugglebug]. The first category relies on instrumenting the system in order to capture objects and other system components at run-time. When a faulty behavior occurs in the field, the stored objects, as well as the entire heap, are

sent to the developers along with the faulty methods to reproduce the crash. These techniques tend to be simple to implement and yield good results, but they suffer from two main limitations. First, code instrumentation comes with a non-negligible overhead on the system. The second limitation is that the collected objects may contain sensitive information causing customer privacy issues. The second category is composed of tools leveraging proprietary data in order to provide hints on potential causes. While these techniques are efficient in improving our comprehension of the bugs, they are not designed with the purpose of reproducing them.

JCHARMING (Java CrasH Automatic Reproduction by directed Model checking) (Mathieu Nayrolles, Hamou-Lhadj, et al. 2015) is a hybrid approach that uses a combination of crash traces and model checking to reproduce bugs that caused field failures automatically. Unlike existing techniques, JCHARMING does not require instrumentation of the code. It does not need access to the content of the heap either. Instead, JCHARMING uses the list of functions, i.e., the crash trace, that are output when an uncaught exception in Java occurs to guide a model checking engine to uncover the statements that caused the crash. Note that a crash trace is sometimes referred to as a stack trace. In this paper, we use these two terms interchangeably.

Model checking (also known as property checking) is a formal technique for automatically verifying a set of properties of finite-state systems (Baier and Katoen 2008). More specifically, this technique builds a graph where each node represents one state of the program and the set of properties that need to be verified in each state. For real-world programs, model checking is often computationally impractical because of the state explosion problem (Baier and Katoen 2008). To address this challenge and apply model checking on large programs, we direct the model checking engine towards the crash using program slicing and the content of the crash trace, and hence, reduce the search space. When applied to reproducing bugs of seven open source systems, JCHARMING achieved 85% accuracy.

The accuracy of JCHARMING, when applied to 30 bugs, is 80%.

8.1 Preliminaries

Model checking (also known as property checking) will, given a formally defined system (that could be software (Visser et al. 2003) or hardware based (Kropf 1999)),

check if the system meets a specification by testing exhaustively all the states of the system under test (SUT), which can be represented by a Kripke (Kripke 1963) structure:

$$SUT = \langle S, S_0, T, L \rangle \quad (5)$$

where S is a set of states, S_0 the set of initial states, T the transitions relations between states and L the labeling function, which labels a state with a set of atomic properties. Figure 33 presents a system with four steps S_0 to S_3 , which have five atomic properties p_1 to p_5 . The labeling function L gives us the properties that are true in a state: $L(S_0) = p_1$, $L(S_1) = p_1, p_2$, $L(S_2) = p_3$, $L(S_3) = p_4, p_5$.

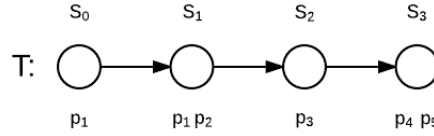


Figure 33: System with four steps S_0 to S_3 , which have five atomic properties p_1 to p_5

The SUT is said to satisfy a property p at a given time if there exists a sequence of states x leading to a state where p holds. This can be written as:

$$(SUT, x) \models p \quad (6)$$

For the SUT of Figure 33, we can write $(SUT, S_0, S_1, S_2) \models p_3$ because the sequence of states S_0, S_1, S_2 will lead to a state S_2 where p_3 holds. However, $(SUT, S_0, S_1, S_2) \models p_3$ only ensures that $\exists x$ such that p is reached at some point in the execution of the program and not that p_3 holds for $\forall x$.

In JCHARMING, we assume that SUTs must not crash in a typical environment. In the framework of this study, we consider a typical environment as any environment where the transitions between the states represent the functionalities offered by the program. For example, in a typical environment, the program heap or other memory spaces cannot be modified. Without this constraint, all programs could be tagged as buggy since we could, for example, destroy objects in memory while the program continues its execution. As we are interested in verifying the absence of unhandled

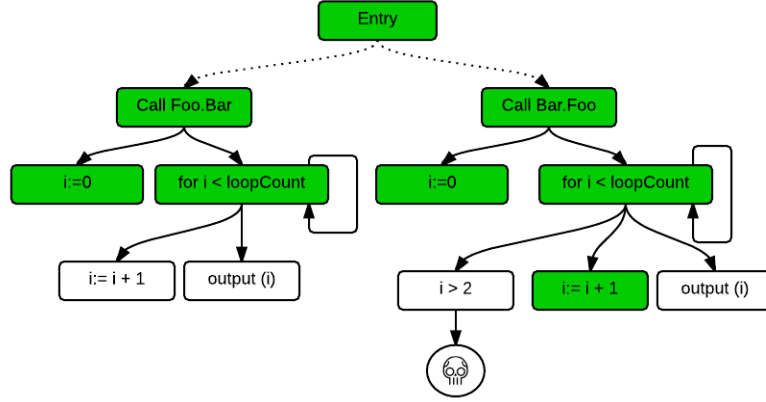


Figure 34: A toy program under testing

exceptions in the SUT, we aim to verify that for all possible combinations of states and transitions there is no path leading towards a crash. That is:

$$\forall x.(SUT, x) \models \neg c \quad (7)$$

If there exists a contradicting path (i.e., $\exists x$ such that $(SUT, x) \models c$) then the model checker engine will output the path x (known as the counter-example), which can then be executed. The resulting Java exception crash trace is compared with the original crash trace to assess if the bug is reproduced. While being accurate and exhaustive in finding counter-examples, model checking suffers from the state explosion problem, which hinders its applicability to large software systems.

To show the contrast between testing and model checking, we use the hypothetical example of Figures 34, 35 and 36 and sketch the possible results of each approach. These figures depict a toy program where from the entry point, unknown calls are made (dotted points) and, at some points, two methods are called. These methods, called `Foo.Bar` and `Bar.Foo`, implement a `for` loop from 0 to `loopCount`. The only difference between these two methods is that the `Bar.Foo` method throws an exception if i becomes larger than two. Hereafter, we denote this property as $p_{i>2}$.

Figure 34 shows the program statements that could be covered using testing approaches. Testing software is a demanding task where a set of techniques is used to test the SUT according to some input. Software testing depends on how well the tester understands the SUT in order to write relevant test cases that are likely to find errors in the program. Program testing is usually insufficient because it is not

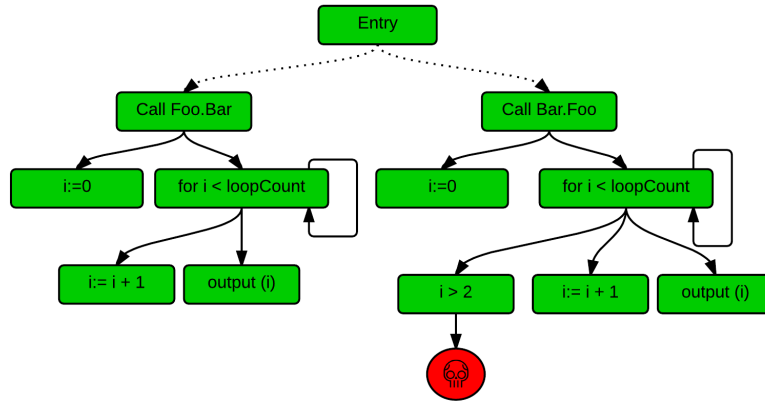


Figure 35: A toy program under model checking

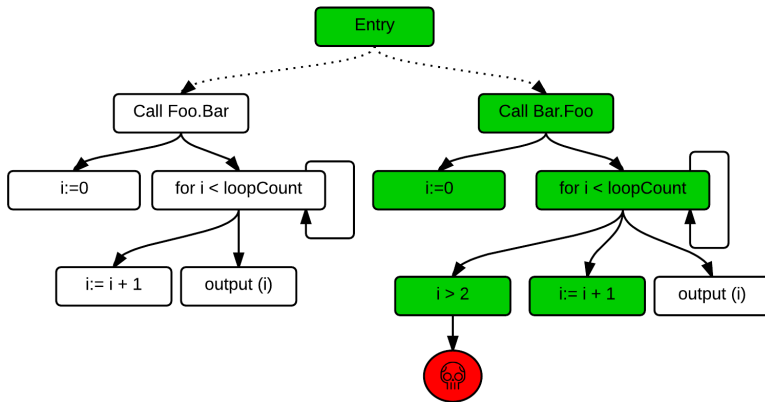


Figure 36: A toy program under directed model checking

exhaustive. In our case, using testing will mean that the tester knows what to look for in order to detect the causes of the failure. We do not assume this knowledge in JCHARMING.

Model checking, on the other hand, explores each and every state of the program (Figure 35), which makes it complete, but impractical for real-world and large systems. To overcome the state explosion problem of model checking, directed (or guided) model checking has been introduced (Edelkamp, Leue, and Lluch-Lafuente 2004; Edelkamp et al. 2009). Directed model checking uses insights — generally heuristics — about the SUT in order to reduce the number of states that need to be examined. Figure 36 explores only the states that may lead to a specific location, in our case, the location of the fault. The challenge, however, is to design techniques that can guide the model checking engine. As we will describe in the next section, we use crash traces and program slicing to overcome this challenge.

Unlike model checking, directed model checking is not complete. In this work, our objective is not to ensure absolute correctness of the program, but to use directed model checking to “hunt” for a bug within the program.

8.2 Approach

Figure 37 shows an overview of JCHARMING. The first step consists of collecting crash traces, which contain raw lines displayed to the standard output when an uncaught exception in Java occurs. In the second step, the crash traces are preprocessed by removing noise (mainly calls to Java standard library methods). The next step is to apply backward slicing using static analysis to expand the information contained in the crash trace while reducing the search space. The resulting slice along with the crash trace are given as input to the model checking engine. The model checker executes statements along the paths from the main function to the first line of the crash trace (i.e., the last method executed at crash time, also called the crash location point). Once the model checker finds inconsistencies in the program leading to a crash, we take the crash stack generated by the model checker and compare it to the original crash trace (after preprocessing). The last step is to build a JUnit test, to be used by software engineers to reproduce the bug in a deterministic way.

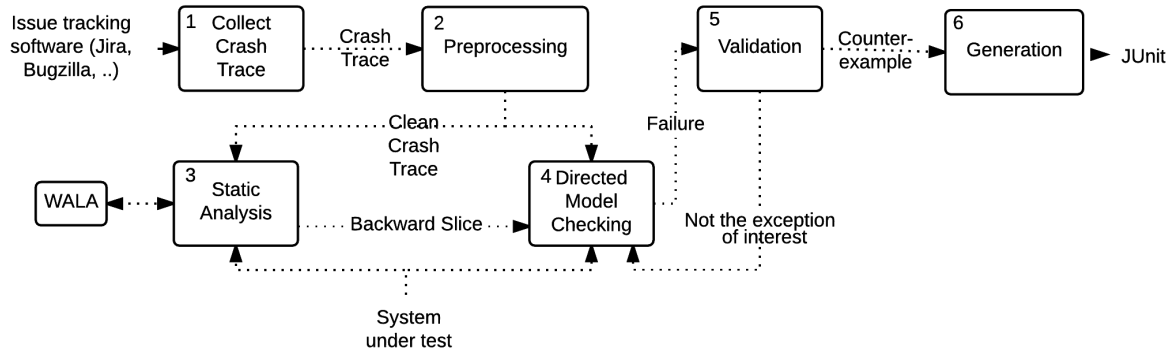


Figure 37: Overview of JCHARMING.

8.2.1 Collecting Crash Traces

The first step of JCHARMING is to collect the crash trace caused by an uncaught exception. Crash traces are usually included in crash reports and can, therefore, be automatically retrieved using a simple regular expression. Figure 38 shows an example of a crash trace that contains the exception thrown when executing the program depicted in Figures 34 to 36. More formally, we define a Java crash trace T of size N as an ordered sequence of frames $T = f_0, f_1, f_2, \dots, f_N$. A frame represents either a method call (with the location of the called method in the source code), an exception, or a wrapped exception. In Figure 38, the frame f_0 represents an exception, the frame f_7 represents a method call to the method `jsep.Foo.buggy`, declared in file `Foo.java` at line 17. In this example, this is the method that caused the exception. f_6 represents a wrapped exception.

It is common in Java to have crash traces that contain wrapped exceptions. Such crash traces are incomplete in the sense that they do not show all the method calls that are invoked from the entry point of the program to the crash point. According to the Java documentation (Oracle 2011), line 8 of Figure 38 should be interpreted as follows: “This line indicates that the remainder of the stack trace for this exception matches the indicated number of frames from the bottom of the stack trace of the exception that was caused by this exception (the “enclosing exception”). This shorthand can greatly reduce the length of the output in the common case where a wrapped exception is thrown from the same method as the “causative exception” is caught.”

We are likely to find shortened traces in bug repositories as they are what the user sees without any possibility to expand their content.

```

1.java.lang.InvalidActivityException:loopTimes
should be < 3
2. at Foo.bar(Foo.java:10)
3. at GUI.buttonActionPerformed(GUI.java:88)
4. at GUI.access$0(GUI.java:85)
5. at GUI$1.actionPerformed(GUI.java:57)
6. caused by java.lang.IndexOutOfBoundsException : 3
7. at jsep.Foo.buggy(Foo.java:17)
8. and 4 more ...

```

Figure 38: Java `InvalidActivityException` is thrown in the `Bar.Foo` loop if the control variable is greater than 2.

In more details, Figure 38 contains a call to the *Bar.foo()* method – the crash location point – and calls to Java standard library functions (in this case, GUI methods because the program was launched using a GUI). As shown in Figure 38, we can see that the first line (referred to as frame f_0 , subsequently the next line is called frame f_1 , etc.) does not represent the real crash point but it is only the last exception of a chain of exceptions. Indeed, the *InvalidActivity* has been triggered by an *IndexOutOfBoundsException* in *jsep.Foo.buggy*. This crash trace shows also an example of nested try-catch blocks.

8.2.2 Preprocessing

In the preprocessing step, we first reconstruct and reorganize the crash trace in order to address the problem of nested exceptions. Nested exception refers to the following structure in Java.

```

1 java.io.IOException: Spill failed
...
14 Caused by: java.lang.IllegalArgumentException
...
28 Caused by: java.lang.NullPointerException

```

In such a case, we want to reproduce the root exception (line 28) that led to the other two (lines 14 and 1). This said, we remove the lines 1 to 14. Then, with the aim to guide the directed model checking engine optimally, we remove frames that

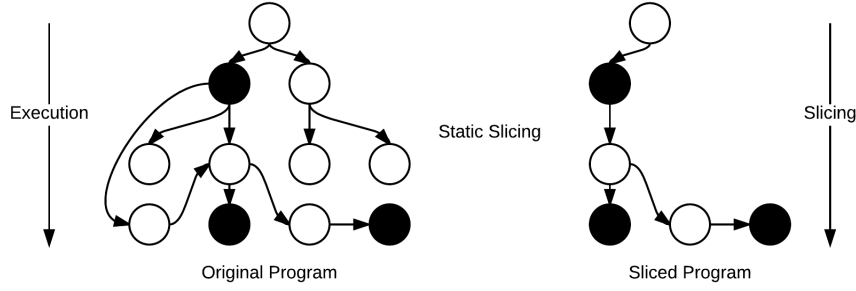


Figure 39: Hypothetical example of static program slicing

are beyond our control. These refer to Java library methods and third party libraries. In Figure 38, we can see that Java GUI and event management components appear in the crash trace. We assume that these methods are not the cause of the crash; otherwise, it means that there is something wrong with the JDK itself. If this is the case, we will not be able to reproduce the crash. Note that removing these unneeded frames will also reduce the search space of the model checker.

8.2.3 Building the Backward Static Slice

Static analysis of programs consists of analyzing programs without executing them or making assumptions about the inputs of the program. There exist many techniques to perform static analysis (data flow analysis, control flow analysis, theorem proving, etc.) that can be used for debugging, program comprehension, and performance analysis. Static slice is a type of static analysis, which takes a program and slicing properties as input in order to create a smaller program with respect to the slicing properties. An example of slicing properties could be to extract only the program statements that modify a certain variable. Static slicing is particularly interesting for JCHARMING as the sliced program, being smaller, will have a smaller state space. We illustrate the process of static slicing using the example in Figure 39. In this figure, we show an example of an abstract syntax tree, generated from a given a program (not shown in this paper). The black states could be states that are impacted by given slicing properties. After the slicing, the final static slice contains the black states and the states that are needed to reach them.

In JCHARMING, we use a particular static slicing known as backward static slicing (De Lucia 2001). A backward slice contains all possible branches that may

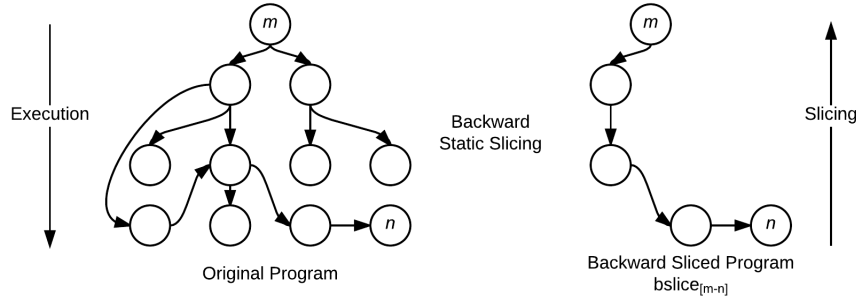


Figure 40: Hypothetical example of backward static program slicing

lead to a point n in the program from a point m as well as the definition of the variables that control these branches (De Lucia 2001). In other words, the slice of a program point n is the program subset that may influence the reachability of point n starting from point m . The backward slice containing the branches and the definition of the variables leading to n from m is noted as $bslice_{[m \leftarrow n]}$. Figure 40 presents an example of a backward static slice.

In JCHARMING, we compute a backward static slice between the location of the crash (which we have in the crash trace) and the entry point of the program (i.e., the main function). However, for large systems, a crash trace does not necessarily contain all the methods that have been executed starting from the entry point of the program to the crash location point. We need to complete the content of the crash trace by identifying all the statements that have been executed starting from the main function until the last line of the preprocessed crash trace. In Figure 38, this will be the function call `Bar.Foo()` which happens to be also the crash location point. To achieve this, we turn to static analyses by extracting a backward slice from the main function of the program to the `Bar.Foo()` method.

$$bslice_{[f_n \leftarrow f_0]} = bslice_{[f_1 \leftarrow f_0]} \cup bslice_{[f_2 \leftarrow f_1]} \cup \dots \cup bslice_{[f_n \leftarrow f_{n-1}]} \quad (8)$$

Note that the union of the slices computed between each pair of frames must be a subset of the final slice between f_0 and the entry point of the program f_n . More formally:

$$\bigcup_{i=0}^{n-1} bslice_{[f_{i+1} \leftarrow f_i]} \subseteq bslice_{[f_n \leftarrow f_0]} \quad (9)$$

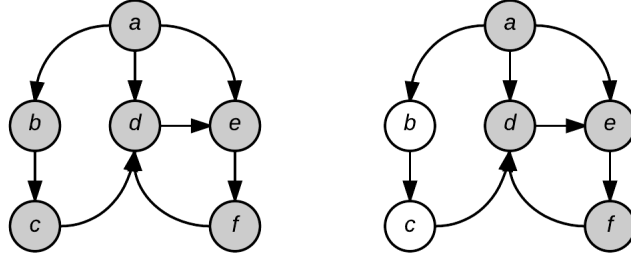


Figure 41: Hypothetical example representing $bslice_{[a \leftarrow d]}$ (left) vs. $\cup_{i=d}^a bslice_{[f_{i+1} \leftarrow f_i]}$ (right) for a crash trace $T = d, f, e$.

Figure 41 presents an example that will help understand Equation 9. In this example, a toy program composed of six methods $a \dots f$ crashes in d with a crash trace $T = \{e, f, d\}$. The backward static slice from the crash point d to the entry point a without using T will be $\{a, b, c, d, e, f\}$ as we do not assume to know the path taken to reach d . However, if we use the information in T to compute the backward static slice, then, some methods can be disregarded. The slice directed by T is $\{a, e, f, d\}$, which is a subset of $\{a, b, c, d, e, f\}$. This example illustrates how a backward slice can be generated using the information in the crash trace.

In the worst case scenario where there exists one and only one transition between each two frames (this is very unlikely for real and complex systems) $bslice_{[entry \leftarrow f_0]}$ and $\cup_{i=0}^n bslice_{[f_{i+1} \leftarrow f_i]}$ yield the same set of states with a comparable computational cost since the number of branches to explore will be the same in both cases.

Algorithm 2 is a high-level representation of how we compute the backward slice between each frame. The algorithm takes as input the preprocessed crash trace $T = \{f_0, f_1, f_2, \dots, f_N\}$, the byte code of the SUT, and the entry point. From line 1 to line 4, we initialize the different variables used by the algorithm. Our algorithm needs to have the crash trace as an array of frames (line 1), the size N of the crash trace (line 2), a `null` backward static slice (line 4, and an offset set to 1 (line 3). The main loop of the algorithm begins at line 5 and ends at line 13. In this loop, we compute the static slice between the current frame and the next one. If the slice is not empty, then we update the final backward slice with the newly computed slice. In Algorithm 1, this is shown as a union between the two slices. Note that this union preserves the order of the elements of the two slices. This can also be seen as a concatenation operation. We describe this process in the subsequent paragraphs

through an example. If the computed slice is empty, it means that Frame $i + 1$ was corrupted then we move to Frame $i + 2$, and so on. At the end of the algorithm, we compute the slice between the last frame and the entry point of the program and update the final slice.

Figure 42 presents a step by step graphical representation of Algorithm 2. In this figure, an hypothetical program, composed of eleven states ($a...k$), crashes at point k . The produced crash trace is composed of six frames $T = \{f_0, f_1, f_2, f_3, f_4, f_5\}$. The frames represent k, i, h, d, b and a , respectively. In the crash trace, f_3 is a corrupt frame and no longer matches a location inside the SUT. This can be the result of a copy-paste error or a deliberate modification made by the reporter of the bug as shown in the case study (see Section 8.4). In such a situation, Algorithm 2 will begin by computing the backward static slice between f_0 (k) and f_1 (i), then between f_1 (i) and f_2 (h). At this point, we passed through the *for* loop (lines 5 to 12) two times, and in both cases the backward static slice was not empty. Consequently, the *if* statement was equal to *true* and we combined both backward static slices in the *bSlice* variable. *bSlice* is equal to $\{k, j, i, h\}$. Then, we want to compute the backward static slice between f_2 (h) and f_3 (d). Unfortunately, f_3 is corrupted and does not point towards a valid location in the SUT. As a result, the slice between f_2 (h) and f_3 (d) will be empty, and we will go to the *else* statement (line 10). Here, we simply increment *offset* by one in order to compute the backward static slice from f_2 (h) and f_4 (b) instead of f_2 (h) and f_3 (d). f_4 is valid and the backward static slice from f_2 (h) and f_4 (b) can be computed and merged to *bSlice*. Finally, we compute the last slice between f_4 (b) and f_5 (a). The final backward static slice is k, i, h, d, b and a .

Our algorithm, given an uncorrupted stack trace, will be able to compute an optimum backward static slice based on the frames (Figure~41) and compensate for frames corruption ,if need be, by *offsetting* the corrupted frame (Figure~42). The compensation of corrupted frames, obviously, comes at the cost of a sub-optimum backward static slice. In our previous example, the transition between b and h could have been omitted if f_3 was not corrupted.

In the rare cases where the final slice is empty (this may happen in situations where the content of the crash trace is seriously corrupted) JCHARMING would simply proceed with non-directed model checking.

Data: Crash Stack, BCode, Entry Point

Result: BSolve

Frame[] frames \leftarrow extract frames from crash stack;

Int n \leftarrow size of frames;

Int offset \leftarrow 1;

Bslice bSlice $\leftarrow \emptyset$;

for $i \leftarrow 0$; ($i < n$ $\&\&$ $offset < n - 1$); $i++$ **do**

 BSlice currentBSlice \leftarrow backward slice from frames[i] to frames[i + offset];

if $currentBSlice \neq \emptyset$ **then**

 bSlice \leftarrow bSlice \cup currentBSlice;

 offset \leftarrow 1;

else

 offset \leftarrow offset + 1;

end

end

Algorithm 2: High-level algorithm computing the union of the slices

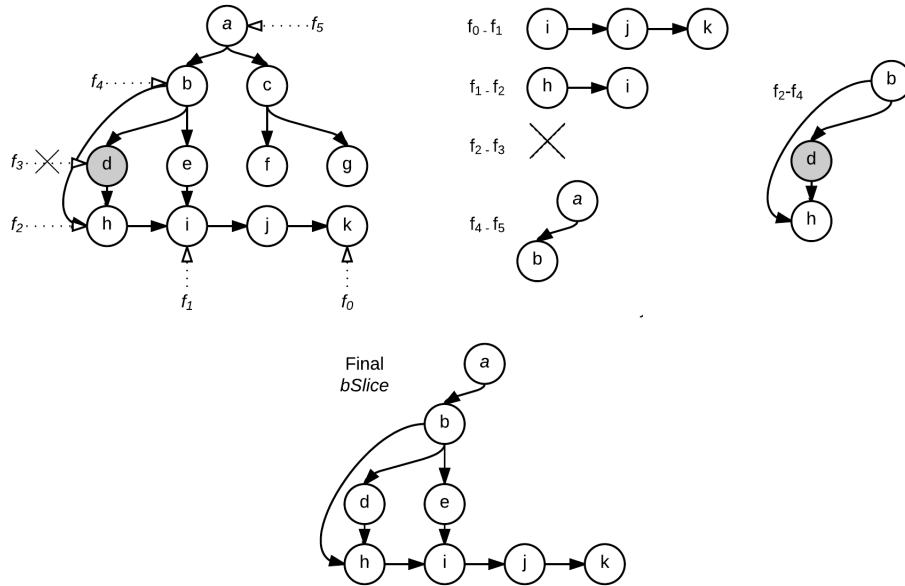


Figure 42: Steps of Algorithm 2 where f_3 is corrupted.

Note that while we allow in JCHARMING the possibility to resort to non-directed model checking, none of the 30 bugs used in this study contained crash traces that were damaged enough for this fall back mechanism to be used.

In order to compute the backward slice, we implement our algorithm as an add-on to the T. J. Watson Libraries for Analysis (WALA) (IBM 2006), which provide static analysis capabilities for Java Byte code and JavaScript. WALA offers a very comprehensive API to perform static backward slicing on Java Byte code from a specific point to another. We did not need to extend WALA to perform our analysis.

Using backward slicing, the search space of the model checker that processes the example of Figures 34 to 36, where a crash happens when $i > 2$, is given by the following expression:

$$Sliced_{SUT} = \left(\begin{array}{c} \bigcup_{i=0}^{entry} bslice_{[f_{i+1} \leftarrow f_i]} \subseteq SUT, \\ S_0, \\ T. \bigcup_{i=0}^{entry} bslice_{[f_{i+1} \leftarrow f_i]} \subseteq T.SUT, \\ L \end{array} \right) \quad (10)$$

Where $\bigcup_{i=0}^{entry} bslice_{[f_{i+1} \leftarrow f_i]} \subseteq SUT$ is the subset of states that can be reached in the computed backward slice, S_0 the set of initial states, $T. \bigcup_{i=0}^{entry} bslice_{[f_{i+1} \leftarrow f_i]}$ the subset of transitions relations between states that exist in the computed backward slice and L the labeling function which labels a state with a set of atomic properties. Then, in the sliced SUT, we try to find:

$$(Sliced_{SUT}, x) \models p_{i>2} \quad (11)$$

That is, there exists a sequence of state transitions x that satisfies $p_{i>2}$. The only frame that needs to be valid for the backward static slice to be meaningful is f_0 . In Figure 38, f_0 is at `Foo.bar(Foo.java : 10)`. If this line of the crash trace is corrupt, then JCHARMING cannot perform the slicing because it does not know where to start the backward slicing. The result is a non-directed model checking, which is likely to fail.

8.2.4 Directed Model Checking

The model checking engine we use in this paper is called JPF (Java PathFinder) (Visser, Psreanu, and Khurshid 2004), which is an extensible JVM for Java byte

code verification. This tool was first created as a front-end for the SPIN model checker (Holzmann 1997) in 1999 before being open-sourced in 2005. The JPF model checker can execute all the byte code instructions through a custom JVM — known as JVM^{JPF}. Furthermore, JPF is an explicit state model checker, very much like SPIN (Holzmann 1997). This is contrasted with a symbolic model checker based on binary decision diagrams (McMillan 1993). JPF designers have opted for a depth-first traversal with backtracking because of its ability to check for temporal liveness properties.

More specifically, JPF’s core checks for defects that can be checked without defining any property. These defects are called *non-functional properties* in JPF and cover deadlock, unhandled exceptions, and **assert** expressions. In JCHARMING, we leverage the *non-functional properties* of JPF as we want to compare the crash trace produced by unhandled exceptions to the crash trace of the bug at hand. In other words, we do not need to define any property ourselves. This said, in JPF, one can define properties by implementing **listeners** — very much like what we did in Section 8.2.6 — that can monitor all actions taken by JPF, which enables the verification of temporal properties for sequential and concurrent Java programs. One of the popular **listeners** of JPF is **jpf-ltl**. This listener supports the verification of method invocations or local and global program variables. **jpf-ltl** can verify temporal properties of method call sequences, linear relations between program variables, and the combination of both. We intend to investigate the use of **jpf-ltl** and the LTL logic to check multi-threaded related crashes as part of future work.

JPF is organized around five simple operations: (i) *generate states*, (ii) *forward*, (iii) *backtrack*, (iv) *restore state* and (v) *check*. In the forward operation, the model checking engine generates the next state s_{t+1} . Each state consists of three distinct components:

- The information of each thread. More specifically, a stack of frames corresponding to method calls.
- The static variables of a given class.
- The instance variables of a given object.

If s_{t+1} has successors, then it is saved in a backtrack table to be restored later. The backtrack operation consists of restoring the last state in the backtrack table. The restore operation allows restoring any state. It can also be used to restore the entire

program as it was the last time we chose between two branches. After each forward, backtrack and restore state operation the check properties operation is triggered.

In order to direct JPF, we have to modify the *generate states* and the *forward* steps. The *generate states* is populated with the states in $\bigcup_{i=0}^{entry} bslice_{[f_{i+1} \leftarrow f_i]} \subset SUT$ and we adjust the *forward step* to explore a state if the target state $s_i + 1$ and the transition x to pass from the current state s_i to s_{i+1} are in $\bigcup_{i=0}^{entry} bslice_{[f_{i+1} \leftarrow f_i]} \subset SUT$ and $x \cdot \bigcup_{i=0}^{entry} bslice_{[f_{i+1} \leftarrow f_i]} \subset x.SUT$.

In other words, JPF does not generate each possible state for the system under test. Instead, JPF generates and explores only the states that fall inside the backward static slice we computed in the previous step. As shown in figure 41, our backward static slice can greatly reduce the space search and is able to compensate for corrupted frames. In short, the idea is that instead of going through all the states of the program as a complete model checker would do, the backward slice directs the model checker to explore only the states that might reach the targeted frame.

8.2.5 Validation

To validate the result of directed model checking, we modify the *check properties* step that checks if the current sequence of state transition x satisfies a set of properties. If the current state transition x can throw an exception, we execute x and compare the exception thrown to the original crash trace (after preprocessing). If the two exceptions match, we conclude that the conditions needed to trigger the failure have been met and the bug is reproduced.

However, as argued by Kim et al. in (Kim et al. 2013), the same failure can be reached from different paths of the program. Although the states executed to reach the defect are not exactly the same, they might be useful to enhance the understanding of the bug by software developers and speed up the deployment of a fix. Therefore, in this paper, we consider a defect to be partially reproduced if the crash trace generated from the model checker matches the original crash trace by a factor of t , where t is a threshold specified by the user. The threshold, t , represents the percentage of identical frames between both crash traces.

For our experiments (see Section 8.4), we set the value of t to 80%. The choice of t should be guided by the need to find the best trade-off between the reproducibility of the bug and the relevance of the generated test cases (the tests should help reproduce

the on-field crash). To determine the best t , we made several incremental attempts, starting from $t = 10\%$. For each attempt, we increased t with a factor of 5% and observed the number of bugs reproduced and the quality of the generated tests. Having $t = 80\%$ provided the best trade-off. This said, we anticipate that the tool that implements our technique should allow software engineers to vary t depending on the bugs and the systems under study. Based on our observations, we recommend, however, to set t to 80% as a baseline. It should also be noted that we deliberately prevented JCHARMING to perform directed model checking with a threshold below 50%. This is because the tests generated with such a low threshold during our experiments did not yield qualitative results.

8.2.6 Generating Test Cases for Bug Reproduction

To help software developers reproduce the crash in a lab environment, we automatically produce the JUnit test cases necessary to run the SUT to cause the bug.

To build a test suite that reproduces a defect, we need to create a set of objects used as arguments for the methods that will enable us to travel from the entry point of the program to the defect location. JPF has the ability to keep track of what happens during model checking in the form of traces containing the visited states and the value of the variables. We leverage this capability to create the required objects and call the methods leading to the failure location.

During the testing of the SUT, JPF emits a trace that is composed of the executed instructions. For large systems, this trace can contain millions of instructions. It is stored in memory and therefore can be queried while JPF is running. However, accessing the trace during the JPF execution considerably slows down the checking process as both the querying mechanism and the JPF engine compete with each other for resources. In order to allow JPF to use 100% of the available resources and still be able to query the executed instructions, we implemented a listener that listens to the JPF trace emitter. Each time JPF processes a new instruction, our listener catches it and saves it into a MongoDB database to be queried in a post-mortem fashion. Figure 43 presents a high-level architecture of the components of the JUnit test generation process.

When the *validate* step triggers a crash stack with a similarity larger than a factor t , the JUnit generation engine queries the MongoDB database and fetches

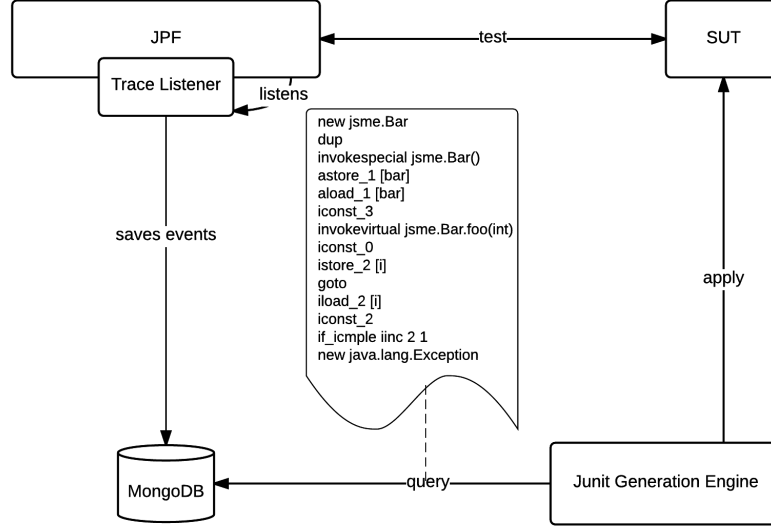


Figure 43: High level architecture of the JUnit test case generation

the sequence of instructions that led to the crash of interest. Figure 43 contains a hypothetical sequence of instructions related to the example of Figures 34, 35, 36, which reads : *new jsme.Bar*, *invokespecial jsme.Bar()*, *astore_1 [bar]*, *aload_1 [bar]*, *iconst_3*, *invirtual jsme.Bar.foo(int)*, *iconst_0*, *istore_2 [i]*, *goto*, *iload_2 [i]*, *iconst_2*, *if_icmple iinc 2 1*, *new java.lang.Exception*. From this sequence we know that, to reproduce to crash of interest, we have to (1) create a new object *jsme.Bar* (*new jsme.Bar*, *invokespecial jsme.Bar()*), (2) store the newly created object in a variable named *bar* (*astore_1 [bar]*), (3) invoke the method *jsme.Bar.foo(int)* of the *bar* object with 3 as value (*aload_1 [bar]*, *iconst_3*, *invirtual jsme.Bar.foo(int)*). Then, the *jsme.Bar.foo(int)* method will execute the *for – loop* from *i = 0* until *i = 3* and throw an exception at *i = 3* (*iconst_0*, *istore_2 [i]*, *goto*, *iload_2 [i]*, *iconst_2*, *if_icmple iinc 2 1*, *new java.lang.Exception*).

The generation of the JUnit test itself is based on templates and targets directly the system under test. Templates are excerpts of Java source code with well-defined tags that will be replaced by values. We use templates because the JUnit test cases have common structures. Figure 44 shows our template for generating JUnit test cases. In this figure, each `{% %}` will be dynamically replaced by the corresponding value when the JUnit test is generated. For example, `{% SUT %}` will be replaced by *Ant* if the SUT is *Ant*. First, we declare four variables that contain the *failure*, the

threshold above which a given bug is said to be partially reproduced, the *differences*, which count how many lines differ between the original failure, the failure produced by JCHARMING, and a *StringTokenizer*. The *StringTokenizer* allows breaking the original *failure* into tokens. Second, the test method where `{%SUT%}` is replaced by the name of the SUT and `{%STEPS%}` by the steps to make the SUT crash. Then, the crash trace related to the crash is received in the *catch* part of the *try-catch* block. In the *catch* part, we compute the number of lines that do not match the original exception and store it into *differences*¹. Finally, the *assertTrue* call will assert that the crash traces from the induced and the original crash are at least *threshold* percent similar to each other.

8.3 Experimental Setup

In this section, we show the effectiveness of JCHARMING to reproduce bugs in ten open source systems. The case studies aim to answer the following question: *Can we use crash traces and directed model checking to reproduce on- field bugs in a reasonable amount of time?*

8.3.1 Targeted Systems

Table 8 shows the systems and their characteristics in terms of Kilo Line of Code (KLoC) and Number of Classes (NoC).

Apache Ant (Apache Software Foundation, n.d.) is a popular command-line tool to build Makefiles. While it is mainly known for Java applications, Apache Ant also allows building C and C++ applications. We choose to analyze Apache Ant because other researchers in similar studies have used it.

ArgoUML (CollabNet, n.d.) is one of the major players among the open source UML modeling tools. It has many years of bug management and, similar to Apache Ant, it has been extensively used as a test subject in many studies.

Dnsjava (Wellington 2013) is a tool for the implementation of the DNS mechanisms in Java. This tool can be used for queries, zone transfers, and dynamic updates. It is not as large as the other two, but it still makes an interesting case subject because it

¹This code has been replaced by `//Countthedifferences` to ease the reading.


```

public class {% SUT %}-{% BUG %} extends TestCase {

    private static final String failure = {% CRASHSTACK
        ↪ %};
    private static final int threshold = {% THRESHOLD %};
    private static int differences = Integer.MAX_VALUE;
    private static final StringTokenizer tokenizerFailure
        ↪ =
        new StringTokenizer(failure , "\n");

    @Test
    public test{% SUT %}() {
        try {
            {% STEPS %}
        } catch (Exception e) {
            // Count the differences
        }
        assertTrue(differences <=
            ↪ tokenizeOriginalFailure
                .countTokens() / 100 * (threshold
                    ↪ -100));
    }
}

```

Figure 44: Simplified Unit Test template

has been well maintained for the past decade. Also, this tool is used in many other popular tools such as Aspirin, Muffin and Scarab.

JfreeChart (Object Refinery Limited 2005) is a well-known library that enables the creation of professional charts. Similar to dnsjava, it has been maintained over a very long period of time. JfreeChart was created in 2005. It is a relatively large application.

Apache Log4j (The Apache Software Foundation 1999) is a logging library for Java. This is not a very large library, but thousands of programs extensively use

Table 8: List of target systems in terms of Kilo line of code (KLoC), number of classes (NoC) and Bug # ID

SUT	KLOC	NoC	Bug #ID
Ant	265	1,233	38622, 41422
ArgoUML	58	1,922	2603, 2558, 311, 1786
dnsjava	33	182	38
jfreechart	310	990	434, 664, 916
Log4j	70	363	11570, 40212, 41186, 45335, 46271, 47912, 47957
MCT	203	1267	440ed48
pdfbox	201	957	1,412, 1,359
Hadoop	308	6,337	2893, 3093, 11878
Mahout	287	1,242	486, 1367, 1594, 1635
ActiveMQ	205	3,797	1054, 2880, 2880
Total	1,517	17,348	30

it. As other Apache projects, this tool is well maintained by a strong open source community and allows developers to submit bugs. The bugs that are in the bug reporting system of Log4j are generally well documented. In addition, the majority of bugs contain crash traces, which makes Log4j a good candidate system for this study.

MCT (NASA 2009) stands for Mission Control technologies and was developed by the NASA Ames Research Center (the creators of JPF) for use in spaceflight mission operation. This tool benefits from two years of history and targets a very critical domain, Spacial Mission Control. Therefore, this tool has to be particularly and carefully tested and, consequently, the remaining bugs should be hard to discover and reproduce.

PDFBox (Apache Software Foundation 2014) is another tool supported by the Apache Software Foundation since 2009 and was created in 2008. PDFBox allows the creation of new PDF documents and the manipulation of existing documents.

Hadoop (Apache Software Foundation 2011) is a framework for storing and processing large datasets in a distributed environment. It contains four main modules: *Common*, *HDFS*, *YARN* and *MapReduce*. In this paper, we study the *Common* module that contains the different libraries required by the other three modules.

Mahout (Apache Software Foundation 2012) is a relatively new software application, built on top of Hadoop. We used Mahout version 0.11, which was released in August 2015. Mahout supports various machine learning algorithms with a focus on collaborative filtering, clustering, and classification.

Finally, ActiveMQ (Snyder, Bosanac, and Davies 2011) is an open source messaging server that allows applications written in Java, C, C++, C#, Ruby, Perl or PHP to exchange messages using various protocols. ActiveMQ has been actively maintained since it became an Apache Software Foundation project in 2005.

8.3.2 Bug Selection and Crash Traces

In this study, we have selected the reproduced bugs randomly in order to avoid the introduction of any bias. We selected a random number of bugs ranging from 1 to 10 for each SUT containing the word “exception” and where the description of the bug contains a match to a regular expression designed to find the pattern of a Java exception.

8.4 Empirical Validation

Table 9 shows the results of JCHARMING in terms of Bug #ID, reproduction status, and execution time (in minutes) of directed model checking (DMC), length of the counter-example (statements in the JUnit test), and execution time (in minutes) for Model Checking (MC). The experiments have been conducted on a Linux machine (8 GB of RAM and using Java 1.7.0_51).

- The result is noted as “Yes” if the bug has been fully reproduced, meaning that the crash trace generated by the model checker is identical to the crash trace collected during the failure of the system.
- The result is “Partial” if the similarity between the crash trace generated by the model checker and the original crash trace is above $t=80\%$ and below $t=100\%$. Given an 80% similarity threshold, we consider partial reproduction as successful. A different threshold could be used.
- Finally, the result of the approach is reported as “No” if either the similarity is below $t < 80\%$ or the model checker failed to crash the system given the input we provided.

Table 9: Effectiveness of JCHARMING using directed model checking (DMC) in minutes, length of the generated JUnit tests (CE length) and model checking (MC) in minutes

SUT	Bug #ID	Reprod.	Time DMC	CE length	Time MC
Ant	38622	Yes	25.4	3	-
	41422	No	42.3	-	-
ArgoUML	2558	Partial	10.6	3	-
	2603	Partial	9.4	3	-
	311	Yes	11.3	10	-
	1786	Partial	9.9	6	-
DnsJava	38	Yes	4	2	23
jFreeChart	434	Yes	27.3	2	-
	664	Partial	31.2	3	-
	916	Yes	26.4	4	-
Log4j	11570	Yes	12.1	2	-
	40212	Yes	15.8	3	-
	41186	Partial	16.7	9	-
	45335	No	3.2	-	-
	46271	Yes	13.9	4	-
	47912	Yes	12.3	3	-
	47957	No	2	-	-
MCT	440ed48	Yes	18.6	3	-
PDFBox	1412	Partial	19.7	4	-
	1359	No	7.5	-	-
Mahout	486	Partial	34.5	5	-
	1367	Partial	21.1	7	-
	1594	No	14.8	-	-
	1635	Yes	31.0	14	-
Hadoop	2893	Partial	7.4	3	32
	3093	Yes	13.1	2	-
	11878	Yes	17.4	6	-
ActiveMQ	1054	Yes	38.3	11	-
	2880	Partial	27.4	6	-
	5035	No	1	-	-

As we can see in Table 9, we were able to reproduce 24 out of 30 bugs either completely or partially (80% success ratio). The average time to reproduce a bug was 19 minutes. The average time in cases where JCHARMING failed to reproduce the bug was 11 minutes. The maximum fail time was 42.3 minutes, which was the time required for JCHARMING to fill all the available memory and stop and a – denotes that JCHARMING reached a sixty-minute timeout. Finally, we report the number of statements in the produced JUnit test, which represents the length of the counter-example. While reproducing a bug is the first step in understanding the cause of a field crash, the steps to reproduce the bug should be as few as possible. It is important for counter-examples to be short to help the developers provide a fix effectively. In average, JCHARMING counter-examples were composed of 5.04 Java statements, which, in our view, is considered reasonable for our approach to be adopted by software developers. This result demonstrates the effectiveness of our approach, more particularly, the use of backward slicing to create a manageable search space that guides the model checking engine adequately. We also demonstrated that our approach is usable in practice since it is also time efficient. Among the 30 different bugs we have tested, we will describe two bugs (chosen randomly) for each category (successfully reproduced, partially reproduced, and not reproduced) for further analysis. The bug report presented in the following sections are the original reports as submitted by the reporter. As such, they contain typos and spelling mistakes that we did not correct.

8.4.1 Successfully Reproduced

The first bug we describe in this discussion is the bug #311 belonging to ArgoUML. This bug was submitted in an earlier version of ArgoUML. This bug is very simple to manually reproduce thanks to the extensive description provided by the reporter, which reads: *I open my first project (Untitled Model by default). I choose to draw a Class Diagram. I add a class to the diagram. The class name appears in the left browser panel. I can select the class by clicking on its name. I add an instance variable to the class. The attribute name appears in the left browser panel. I can't select the attribute by clicking on its name. Exception occurred during event dispatching:* The reporter also attached the crash trace presented in Figure 45 that we used as input for JCHARMING:

```

1. java.lang.NullPointerException:
2. at
3. uci.uml.ui.props.PropPanelAttribute .setTargetInternal (PropPanelAttribute.java)
4. at uci.uml.ui.props.PropPanel. setTarget(PropPanel.java)
5. at uci.uml.ui.TabProps.setTarget(TabProps.java)
6. at uci.uml.ui.DetailsPane.setTarget (DetailsPane.java)
7. at uci.uml.ui.ProjectBrowser.select (ProjectBrowser.java)
8. at uci.uml.ui.NavigatorPane.mySingleClick (NavigatorPane.java)
9.          at      uci.uml.ui.NavigatorPane$Navigator      MouseListener.mouse
Clicked(NavigatorPane.java)
10.at      java.awt.AWTEventMulticaster.mouseClicked      (AWTEventMulticas-
ter.java:211)
11.      at java.awt.AWTEventMulticaster.mouseClicked (AWTEvent  Multicast
er.java:210)
12.at java.awt.Component.processMouseEvent (Component.java:3168)
...
19. java.awt.LightweightDispatcher .retargetMouseEvent (Container.java:2068)
22. at java.awt.Container .dispatchEventImpl (Container.java:1046)
23. at java.awt.Window .dispatchEventImpl (Window.java:749)
24. at java.awt.Component .dispatchEvent (Component.java:2312)
25. at java.awt.EventQueue .dispatchEvent (EventQueue.java:301)
28. at java.awt.EventDispatchThread.pumpEvents
(EventDispatch Thread.java:90)
29. at java.awt.EventDispatchThread.run(EventDispatch Thread.java:82)

```

Figure 45: Crash trace reported for bug ArgoUML #311

The cause of this bug is that the reference to the attribute of the class was lost after being displayed on the left panel of ArgoUML and therefore, selecting it through a mouse click throws a null pointer exception. In the subsequent version, ArgoUML developers added a TargetManager to keep the reference of such object in the program. Using the crash trace, JCHARMING’s preprocessing step removed the lines between lines 11 and 29 because they belong to the Java standard library and we do not want either the static slice or the model checking engine to verify the Java standard library but only the SUT. Then, the third step performs the static analysis following the process described in Section IV.C. The fourth step performs the model checking on the static slice to produce the same crash trace. More specifically, the model checker identifies that the method *setTargetInternal(Object o)* could receive a null object that will result in a *Null* pointer exception.

The second reproduced bug we describe in this section is Bug #486 belonging to MAHOUT. The submitter (Robin Anil) named the bug entry as *Null Pointer Exception running DictionaryVectorizer with ngram=2 on Reuters dataset*. He simply copied the crash stack presented in Figure~46 without further explanation.

Drew Farris², who was assigned to fix this bug, commented *Looks like this was due to an improper use of the Gram default constructor that happened as a part of the 0.20.2³ refactoring work*. While this quick comment, made only two and a half hours after the bug submission, was insightful as shown in our generated test case⁴, the fix happened in the *CollocCombiner* class that is one of the *Reducer*⁵ available in Mahout. The fix (commit #f13833) involved creating an iterator to combine the frequencies of the *Gram* and a null check of the final frequency.

JCHARMING’s preprocessing step removed the lines between lines 1 to 14 because they belong to the second thrown exception since *java.lang.NullPointerException* occurred when writing in a *ByteArrayOutputStream*. JCHARMING aims to reproduce the root exceptions and not the exceptions that derive from other exceptions. This said, the *java.io.IOException: Spill failed* was ignored, and our directed model checking engine focused, with success, on reproducing the *java.lang.NullPointerException*.

²<https://issues.apache.org/jira/browse/MAHOUT-486>

³Farris certainly meant 0.10.2 which was the last refactor of the incriminated class, and the current version of Mahout is 0.11

⁴As a reminder, the generated test cases are made available at research.mathieu-nayrolles.com/jcharming

⁵As in Map/Reduce

```
1 java.io.IOException: Spill failed
2 at org.apache.hadoop.mapred.MapTask$MapOutputBuffer.collect(MapTask.java:860)
...
14 Caused by: java.lang.NullPointerException
15 at java.io.ByteArrayOutputStream.write(ByteArrayOutputStream.java:86)
16 at java.io.DataOutputStream.write(DataOutputStream.java:90)
17 at org.apache.mahout.utils.nlp.collocations.llr.Gram.write(Gram.java:181)
18 at org.apache.hadoop.io.serializer.WritableSerialization$WritableSerializer.serialize
...
19 at org.apache.hadoop.io.serializer.WritableSerialization$WritableSerializer.serialize
...
20 at org.apache.hadoop.mapred.IFile$Writer.append(IFile.java:179)
21 at org.apache.hadoop.mapred.Task$CombineOutputCollector.collect(Task.java:880)
22 at org.apache.hadoop.mapred.Task$NewCombinerRunner$OutputConverter.write
...
23 at org.apache.hadoop.mapreduce.TaskInputOutputContext.write ...
24 at org.apache.mahout.utils.nlp.collocations.llr.CollocCombiner.reduce(CollocCombiner.java:40)
25 at org.apache.mahout.utils.nlp.collocations.llr.CollocCombiner.reduce(CollocCombiner.java:25)
26 at org.apache.hadoop.mapreduce.Reducer.run(Reducer.java:176)
27 at org.apache.hadoop.mapred.Task$NewCombinerRunner.combine(Task.java:1222)
28 at org.apache.hadoop.mapred.MapTask$MapOutputBuffer.sortAndSpill(MapTask.java:1265)
29 at org.apache.hadoop.mapred.MapTask$MapOutputBuffer.access$1800(MapTask.java:686)
30 at org.apache.hadoop.mapred.MapTask$MapOutputBuffer$SpillThread.run(MapTask.java:1173)
```

Figure 46: Crash trace reported for bug Mahout #486


```

1. java.lang.IllegalArgumentException: null source
2. at java.util.EventObject.<init> (EventObject.java:38)
3. at
4 org.jfree.chart.ChartMouseEvent.<init> (ChartMouseEvent.java:83)
5. at org.jfree.chart.ChartPanel .mouseMoved(ChartPanel.java:1692)
6. <deleted entry>

```

Figure 47: Crash trace reported for bug JFreeChart #664

8.4.2 Partially Reproduced

As an example of a partially reproduced bug, we explore Bug #664 of the Jfreechart program. The description provided by the reporter is: *In ChartPanel.mouseMoved there's a line of code which creates a new ChartMouseEvent using as first parameter the object returned by getChart(). For getChart() is legal to return null if the chart is null, but ChartMouseEvent's constructor calls the parent constructor which throws an IllegalArgumentException if the object passed in is null.*

The reporter provided the crash trace containing 42 lines and then replaced an unknown number of lines by the following statement `<$deleted entry$>`. While JCHARMING successfully reproduced a crash yielding almost the same trace as the original trace, the `<$deleted entry$>` statement – which was surrounded by calls to the standard Java library – was not suppressed and stayed in the crash trace. That is, JCHARMING produced only the 6 (out of 7) first lines and reached 83% similarity, and thus a partial reproduction.

The second partially reproduced bug we present here is Bug #2893 belonging to Hadoop. This bug, reported in February 2008 by Lohit Vijayarenu, was titled *checksum exceptions on trunk* and contained the following description: *While running jobs like Sort/WordCount on trunk I see few task failures with ChecksumException. Re-running the tasks on different nodes succeeds. Here is the stack*

```

1 Map output lost , rescheduling: getMapOutput(
    ↪ task_200802251721_0004_m_000237_0,29) failed :
2 org.apache.hadoop.fs.ChecksumException: Checksum error: /
    ↪ tmps/4/apred-tt/mapred-local/
    ↪ task_200802251721_0004_m_000237_0/file.out at 2085376

```

```

3  at org.apache.hadoop.fs.FSInputChecker.verifySum(
    ↪ FSInputChecker.java:276)
4  at org.apache.hadoop.fs.FSInputChecker.readChecksumChunk(
    ↪ FSInputChecker.java:238)
5  at org.apache.hadoop.fs.FSInputChecker.read1(FSInputChecker
    ↪ .java:189)
6  at org.apache.hadoop.fs.FSInputChecker.read(FSInputChecker.
    ↪ java:157)
7  at java.io.DataInputStream.read(DataInputStream.java:132)
8  at org.apache.hadoop.mapred.TaskTracker$MapOutputServlet.
    ↪ doGet(TaskTracker.java:2299)
...
23 at org.mortbay.util.ThreadPool$PoolThread.run(ThreadPool.
    ↪ java:534)

```

Similarly to the first partially reproduced bug, the crash traces produced by our directed model checking engine and the related test case did not match 100% of the attached crash stack. While JCHARMING successfully reproduced the bug, the crash stack contains timestamps information (e.g., 200802251721), that was logically different in our produced stack trace as we ran the experiment years later.

In all bugs that were partially reproduced, we found that the differences between the crash trace generated from the model checker and the original crash trace (after preprocessing) consist of a few lines only.

8.4.3 Not Reproduced

To conclude the discussion on the case study, we present a case where JCHARMING was unable to reproduce the failure. For the bug #47957, belonging to LOG4J and reported in late 2009 the author wrote: *Configure SyslogAppender with a Layout class that does not exist; it throws a NullPointerException. Following is the exception trace:* and attached the following crash trace:

```

1. 10052009 01:36:46 ERROR [Default: 1]
struts.CPExceptionHandler.execute
RID[( null;25KbxlK0voima4h00ZLBQFC;236A18E60000045C3A

```

```

7D74272C4B4A61)]
2. Wrapping Exception in ModuleException
3. java.lang.NullPointerException
4. at org.apache.log4j.net.SyslogAppender
.append(SyslogAppender.java:250)
5. at org.apache.log4j.AppenderSkeleton
.doAppend(AppenderSkeleton.java:230)
6. at org.apache.log4j.helper.AppenderAttachableImpl
.appendLoopOnAppenders(AppenderAttachableImpl
.java:65)
7. at org.apache.log4j.Category.callAppenders
(Category.java:203)
8. at org.apache.log4j.Category
.forcedLog(Category.java:388)
9. at org.apache.log4j.Category.info
(Category.java:663)

```

The first three lines are not produced by the standard execution of the SUT but by an `ExceptionHandler` belonging to Struts (Apache Software Foundation 2000). Struts is an open source MVC (Model View Controller) framework for building Java web applications. JCHARMING examined the source code of Log4J for the crash location *struts.CPEExceptionHandler.execute* and did not find it since this method belongs to the source base of Struts – which uses log4j as a logging mechanism. As a result, the backward slice was not produced, and we failed to perform the next steps. It is noteworthy that the bug is marked as a duplicate of the bug #46271 which contains a proper crash trace. We believe that JCHARMING could have successfully reproduced the crash if it was applied to the original bug.

The second bug that we did not reproduce and that we present in this section belongs to Mahout. Jaehoon Ko reported it on July 2014. Bug #1594 is titled *Example factorize-movielens-1M.sh does not use HDFS* and reads *It seems that factorize-movielens-1M.sh does not use HDFS at all. All paths look local paths, not HDFS. So the example crashes because it cannot find input data from HDFS:*

```

1 Exception in thread $$main'' org.apache.hadoop.mapreduce.
  ↳ lib.input.InvalidInputException: Input path does not
  ↳ exist: /tmp/mahout-work-hoseog.lee/movielens/ratings.
  ↳ csv
2 at org.apache.hadoop.mapreduce.lib.input.FileInputFormat.
  ↳ singleThreadedListStatus ...
3 at org.apache.hadoop.mapreduce.lib.input.FileInputFormat.
  ↳ listStatus ...
...
31 at org.apache.hadoop.util.RunJarm.theain(RunJar.java:212)

```

This entry was marked as *Not A Problem / WONT_FIX* meaning that the reported bug was not a bug in the first place. The resolution of this bug⁶ involved the modification of a bash script that Ko (the submitter) was using to *query* Mahout. In other words, the cause of the failure was external to Mahout itself, and this is why JCHARMING could not reproduce it.

8.5 Threats to Validity

The selection of SUTs is one of the common threats to validity for approaches aiming to improve the understanding of a program's behavior. It is possible that the selected programs share common properties that we are not aware of and therefore, invalidate our results. However, the SUTs analyzed by JCHARMING are the same as the ones used in similar studies. Moreover, the SUTs vary in terms of purpose, size and history.

Another threat to validity lies in the way we have selected the bugs used in this study. We selected the bugs randomly to avoid any bias. One may argue that a better approach would be to select bugs based on complexity or other criteria (severity, etc.). We believe that a complex bug (if complexity can at all be measured) may perhaps have an impact on the running time of the approach, but we are not convinced that the accuracy of our approach depends on the complexity or the type of bugs we use. Instead, it depends on the quality of the produced crash trace. This said, in theory, we may face situations where the crash trace is completely corrupted. In such cases, there is nothing that guides the model checker. In other words, we will end up running

⁶<https://github.com/apache/mahout/pull/38#issuecomment-51436303>

a full model checker. It is difficult to evaluate the number of times we may face this situation without conducting an empirical study on the quality of crash traces. We defer this to future work.

In addition, we see a threat to validity that stems from the fact that we only used open source systems. The results may not be generalizable to industrial systems.

Field failures can also occur due to the running environment in which the program is executed. For instance, the failure may have been caused by the reception of a network packet or the opening of a given file located on the hard drive of the users. The resulting failures will hardly be reproducible by JCHARMING.

Finally, the programs we used in this study are all written in the Java programming language and JCHARMING leverages the crash traces produced by the JVM to reproduce bugs. This can limit the generalization of the results. However, similar to Java, .Net, Python and Ruby languages also produce crash traces. Therefore, JCHARMING could be applied to other object-oriented languages.

In conclusion, internal and external validity have both been minimised by choosing a relatively large set of different systems and using input data that can be found in other programming languages.

8.6 Chapter Summary

In this chapter, we presented JCHARMING (Java CrasH Automatic Reproduction by directed Model checking), an automatic bug reproduction technique that combines crash traces and directed model checking. JCHARMING relies on crash traces and backward program slices to direct a model checker. This way, we do not need to visit all the states of the subject program, which would be computationally taxing. When applied to thirty bugs from ten open source systems, JCHARMING was able to successfully reproduce 80% of the bugs. The average time to reproduce a bug was 19 minutes, which is quite reasonable, given the complexity of reproducing bugs that cause field crashes.

This said, JCHARMING suffers from three main limitations. The first one is that JCHARMING cannot reproduce bugs caused by multi-threading. We can overcome this limitation by using advanced features of the JPF model checker such as the *jpf-ltl* listener. The *jpf-ltl* listener was designed to check temporal properties of concurrent

Java programs. The second limitation is that JCHARMING cannot be used if external inputs cause the bug. We can always build a monitoring system to retrieve this data, but this may lead to privacy concerns. Finally, the third limitation is that the performance of JCHARMING relies on the quality of the crash traces. This limitation can be addressed by investigating techniques that can improve the reporting of crash traces. For the time being, the bug reporters simply copy and paste (and modify) the crash traces into the bug description. A better practice would be to automatically append the crash trace to the bug report, for example, in a different field than the bug description.

This chapter concludes our work targeted to software maintenance. In the next chapter, we propose a taxonomy of bugs that aims to categorize the research in this area.

Chapter 9

Towards a Classification of Bugs Based on the Location of the Corrections: An Empirical Study

9.1 Introduction

In the previous chapters, we investigated how to prevent clones and defect, proposes fixes at commit-time and reproduce field-crash.

Our works (Nayrolles et al. 2016; Nayrolles and Hamou-Lhadj 2016; Maiga et al. 2015; M. Nayrolles et al. 2015; Nayrolles and Hamou-lhadj 2018; Mathieu Nayrolles, Hamou-Lhadj, et al. 2015)) treat bug as equal in a sense that they do not assume any underlying classification of bugs. It also the case for other studies. By a fix, we mean a modification (adding or deleting lines of code) to an existing file that is used to solve the bug.

For example, there have been several studies (e.g., (Wei, Zimmermann, and Zeller 2007; Zhang, Gong, and Versteeg 2013)) that study the factors that influence the bug fixing time. These studies empirically investigate the relationship between bug report attributes (description, severity, etc.) and the fixing time. Other studies take bug analysis to another level by investigating techniques and tools for bug prediction and reproduction (e.g., [N. Chen (2013a); Kim, Zimmermann, Whitehead Jr., et al. (2007a);]).

With this in mind, the relationship between bugs and fixes can be modelled using

the UML diagram in Figure 48. The diagram only includes bug reports that are fixed and not, for example, duplicate reports. From this figure, we can think of four instances of this diagram, which we refer to as bug taxonomy or simply bug types (see Figure 49).



Figure 48: Class diagram showing the relationship between bugs and fixed

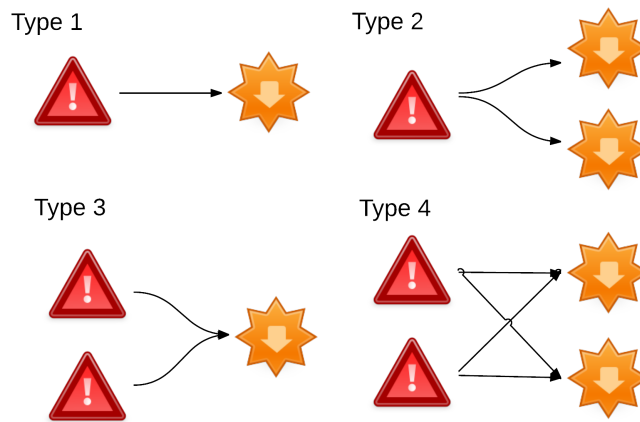


Figure 49: Proposed Taxonomy of Bugs

The first and second types are the ones we intuitively know about. T1 refers to a bug being fixed in one single location (i.e., one file), while T2 refers to bugs being fixed in more than one location. In Figure 2, only two locations are shown for the sake of clarity, but many more locations could be involved in the fix of a bug. T3 refers to multiple bugs that are fixed in the same location. T4 is an extension of T3, where multiple bugs are resolved by modifying the same set of locations. Note that T3 and T4 bugs are not duplicates, they may occur when different features of the system fail due to the same root causes (faults).

In our dataset, composed of 388 projects and presented in section 9.2.1, the proportion of each type of bug is as follows: T1 6.8 %, T2 3.7 %, T3 28.3 % and T4 61.2%. Also, classical measures of complexity such as duplication, fixing time, number of comments, number of time a bug is reopened, files impacted, severity, changesets, hunks, and chunks, also presented in section 9.2.1, show that type 4 are significantly more complex than types 1, 2 and 3.

The existence of a high number of T4 bugs and the fact that they are more complex call for techniques that can effectively tag bug report as T4 at submission time for enhanced triaging. More particularly, we are interested in the following research questions:

- **RQ1:** *Are T4 bug predictable at submission time?* In this research question, we investigate if and how to predict the type of a bug report at submission time. Being able to build accurate classifiers predicting the bug type at submission time will allow improving the triaging and the bug handling process.
- **RQ2:** *What are the best predictors of type 4 bugs ?* This second research question aims to investigate what are the markups that allow for accurate prediction of type 4 bugs.

Our objective is to propose a classification that can allow researchers in the field of mining bug repositories to use the taxonomy as a new criterion in triaging, prediction, and reproduction of bugs. By analogy, we can look at the proposed bug taxonomy similarly as the clone taxonomy presented by Kapser and Godfrey (Kapser and Godfrey 2003). The authors proposed seven types of source code clones and then conducted a case study, using their classification, on the file system module of the Linux operating system. This clone taxonomy continues to be used by researchers to build better approaches for detecting a given clone type and being able to compare approaches with each other effectively. Moreover, we build upon this proposed classification and predict the type of incoming bugs with a 65.40% precision 94.16% recall for f_1 measure of 77.19%.

9.2 Experimental Setup

In this section, we present our datasets in length. In this presentation, we explore the proportion of each type of bug as well as the complexity of each type.

9.2.1 Context Selection

The context of this study consists of the change history of 388 projects belonging to two software ecosystems, namely, Apache and Netbeans. Table 10 reports, for each

of them, (i) the number of resolved-fixed reports, (ii) the number of commits, (iii) the overall number of files, and (iv) the number of projects analysed.

Dataset	R/F BR	CS	Files	Projects
Netbeans	53,258	122,632	30,595	39
Apache	49,449	106,366	38,111	349
Total	102,707	229,153	68,809	388

Table 10: Datasets

All the analysed projects are hosted in *Git* or *Mercurial* repositories and have either a *Jira* or a *Bugzilla* issue tracker associated with them. The Apache ecosystem consists of 349 projects written in various programming languages (C, C++, Java, Python, ...) and uses *Git* with *Jira*. These projects represent the Apache ecosystem in its entirety. We did not exclude any system from our study. The complete list can be found online¹. The Netbeans ecosystem consists of 39 projects, mostly written in Java. Similar to the Apache ecosystem, we selected all the projects belonging to the Netbeans ecosystem. The Netbeans community uses *Bugzilla* with *Mercurial*.

The choice of these two ecosystems is driven by the motivation to consider projects are having (i) different sizes, (ii) different architectures, and (iii) different development bases and processes. Apache projects vary significantly in terms of the size of the development team, purpose and technical choices (Bavota et al. 2013). On the other side, Netbeans has a relatively stable list of core developers and a common vision shared by the 39 related projects (Wang, Baik, and Devanbu 2011).

Cumulatively, these datasets span from 2001 to 2014. In summary, our consolidated dataset contains 102,707 closed issues, 229,153 changesets, 68,809 files that have been modified to fix the bugs, 462,848 comments, and 388 distinct systems. We also collected 221 million lines of code modified to fix bugs, identified 3,284 sub-projects, and 17,984 unique contributors to these bug reports and source code version management systems. The cumulated opening time for all the bugs reaches 10,661 working years (3,891,618 working days). The raw size of the cloned source code alone, excluding binaries, images, and other non-text files, is 163 GB.

¹<https://projects.apache.org/projects.html?name>

Table 11: Contingency table and Pearson’s chi-squared tests

Ecosystem	T1	T2	T3	T4	Pearson’s chi-squared p-Value
Apache	1968 (14.3 %)	1248 (9.1 %)	3101 (22.6 %)	7422 (54 %)	<0.01
Netbeans	776 (2.9 %)	240 (0.9 %)	8372 (31.3 %)	17366 (64.9 %)	
Overall	2744 (6.8 %)	1488 (3.7 %)	11473 (28.3 %)	24788 (61.2 %)	

To assign commits to issues, we used the regular expression based approach proposed by Fischer et al. (Fischer, Pinzger, and Gall, n.d.), which matches the issue ID in the commit note to the commit. Using this technique, we were able to link almost 40% (40,493 out of 102,707) of our resolved/fixed issues to 229,153 commits. Note that an issue can be fixed using several commits.

9.2.2 Dataset Analysis

Using our dataset, we extracted the files f_i impacted by each commit c_i for each one of our 388 projects. Then, we classified the bugs according to each type, which we formulate as follows:

- **Type 1:** A bug is tagged T1 if it is fixed by modifying a file f_i , and f_i is not involved in any other bug fix.
- **Type 2:** A bug is tagged T2 if it is fixed by modifying by n files, $f_{i..n}$, where $n > 1$, and the files $f_{i..n}$ are not involved in any other bug fix.
- **Type 3:** A bug is tagged T3 if it is fixed by modifying a file f_i and the file f_i is involved in fixing other bugs.
- **Type 4:** A bug is tagged T4 if it is fixed by modifying several files $f_{i..n}$ and the files $f_{i..n}$ are involved in any other bug fix.

Table 11 presents the contingency table and the results of the Pearson’s chi-squared tests we performed on each type of bug. We can see that the proportion of T4 (61.2%) largely higher than that of T1 (6.8%), 2 (3.7%) and 3 (28.3%) and that the difference is significant according to the Pearson’s chi-squared test.

Pearson’s chi-squared independence test is used to analyse the relationship between two qualitative data, in our study the type bugs and the studied ecosystem. The results of Pearson’s chi-square independence tests are considered statistically significant at $\alpha = 0.05$. If $p\text{-value} \leq 0.05$, we can conclude that the proportion of each type is significantly different.

We analyse the complexity of each bug regarding duplication, fixing time, number of comments, number of time a bug is reopened, files impacted, severity, changesets, hunks, and chunks.

Complexity metrics are divided into two groups: (a) process and (b) code metrics. Process metrics refer to metrics that have been extracted from the project tracking system (i.e., fixing time, comments, reopening and severity). Code metrics are directly computed using the source code used to fix a given bug (i.e., files impacted, changesets required, hunks and chunks). We acknowledge that these complexity metrics only represent an abstraction of the actual complexity of a given bug as they cannot account for the actual thought process and expertise required to craft a fix. However, we believe that they are an accurate abstraction. Moreover, they are used in several studies in the field to approximate the complexity of a bug (Wei, Zimmermann, and Zeller 2007; Saha, Khurshid, and Perry 2014; Nam, Pan, and Kim 2013; Anvik, Hiew, and Murphy 2006; N. Nagappan and Ball 2005).

Tables 12, 13 and 14 present descriptive statistics about each metric for each bug type per ecosystem and for both ecosystems combined. The descriptive statistics used are μ :mean, \sum :sum, \hat{x} :median, σ :standard deviation and %:percentage. We also show the results of Mann-Whitney test for each metric and type. We added the \checkmark symbol to the Mann-Whitney tests results columns when the value is statistically significant (e.g. $\alpha < 0.05$) and \times otherwise.

Duplicate

The duplicate metric represents the number of times a bug gets resolved using the *duplicate* label while referencing one of the *resolved/fixed* bug of our dataset. The process metric is useful to approximate the impact of a given bug on the community. For a bug to be resolved using the *duplicate*, it means that the bug has been reported before. The more a bug gets reported by the community, the more people are impacted enough to report it. Note that, for a bug_a to be resolved using the *duplicate* label and referencing bug_b, bug_b does not have to be resolved itself. Indeed, bug_b could be under investigation (i.e. *unconfirmed*) or being fixed (i.e. *new* or *assigned*). Automatically detecting duplicate bug report is a very active research field (Sun et al. 2011; Bettenburg, Premraj, and Zimmermann 2008; A. T. Nguyen et al. 2012; Jalbert and Weimer 2008; Tian, Sun, and Lo 2012; Runeson, Alexandersson, and

Table 12: Apache Ecosystem Complexity Metrics Comparison and Mann-whitney test results.

μ :mean,

\sum :sum, \hat{x} :median, σ :standard deviation, %:percentage

Types	Metric	μ	\sum	\hat{x}	σ	%	T1	T2	T3	T4
T1	Dup.	0.026	51	0	0.2	14.8	n.a	$\mathbf{X}(0.53)$	$\check{(<0.05)}$	$\mathbf{X}(0.45)$
	Tim.	91.574	180217	4	262	21.8	n.a	$\check{(<0.05)}$	$\check{(<0.05)}$	$\check{(<0.05)}$
	Com.	4.355	8571	3	4.7	9.5	n.a	$\check{(<0.05)}$	$\mathbf{X}(0.17)$	$\check{(<0.05)}$
	Reo.	0.062	122	0	0.3	13.8	n.a	$\mathbf{X}(0.29)$	$\check{(<0.05)}$	$\check{(<0.05)}$
	Fil.	0.991	1950	1	0.1	3.7	n.a	$\check{(<0.05)}$	$\mathbf{X}(0.28)$	$\check{(<0.05)}$
	Sev.	3.423	6737	4	1.3	13.2	n.a	$\mathbf{X}(0.18)$	$\check{(<0.05)}$	$\check{(<0.05)}$
	Cha.	1	1968	1	0	1.9	n.a	$\check{(<0.05)}$	$\check{(<0.05)}$	$\check{(<0.05)}$
	Hun.	3.814	7506	3	2.4	0	n.a	$\check{(<0.05)}$	$\check{(<0.05)}$	$\check{(<0.05)}$
	Chur.	18.761	36921	7	48.6	0	n.a	$\check{(<0.05)}$	$\mathbf{X}(0.09)$	$\check{(<0.05)}$
T2	Dup.	0.022	28	0	0.1	8.1	$\mathbf{X}(0.53)$	n.a	$\mathbf{X}(0.16)$	$\mathbf{X}(0.19)$
	Tim.	115.158	143717	8	294.1	17.4	$\check{(<0.05)}$	n.a	$\check{(<0.05)}$	$\check{(<0.05)}$
	Com.	5.041	6291	4	4.7	7	$\check{(<0.05)}$	n.a	$\check{(<0.05)}$	$\check{(<0.05)}$
	Reo.	0.071	89	0	0.3	10.1	$\mathbf{X}(0.29)$	n.a	$\check{(<0.05)}$	$\mathbf{X}(0.59)$
	Fil.	4.381	5468	2	20.4	10.5	$\check{(<0.05)}$	n.a	$\check{(<0.05)}$	$\check{(<0.05)}$
	Sev.	3.498	4365	4	1.2	8.6	$\mathbf{X}(0.18)$	n.a	$\check{(<0.05)}$	$\check{(<0.05)}$
	Cha.	4.681	5842	2	20.4	5.5	$\check{(<0.05)}$	n.a	$\check{(<0.05)}$	$\check{(<0.05)}$
	Hun.	561.995	701370	14	13628.2	3.9	$\check{(<0.05)}$	n.a	$\check{(<0.05)}$	$\check{(<0.05)}$
	Chur.	14184.869	17702716	88	400710.2	8	$\check{(<0.05)}$	n.a	$\check{(<0.05)}$	$\check{(<0.05)}$
T3	Dup.	0.016	50	0	0.1	14.5	$\check{(<0.05)}$	$\mathbf{X}(0.16)$	n.a	$\check{(<0.05)}$
	Tim.	35.892	111300	1	151.8	13.5	$\check{(<0.05)}$	$\check{(<0.05)}$	n.a	$\check{(<0.05)}$
	Com.	4.422	13712	3	4.4	15.2	$\mathbf{X}(0.17)$	$\check{(<0.05)}$	n.a	$\check{(<0.05)}$
	Reo.	0.033	101	0	0.2	11.5	$\check{(<0.05)}$	$\check{(<0.05)}$	n.a	$\check{(<0.05)}$
	Fil.	0.994	3081	1	0.1	5.9	$\mathbf{X}(0.28)$	$\check{(<0.05)}$	n.a	$\check{(<0.05)}$
	Sev.	3.644	11300	4	1.1	22.2	$\check{(<0.05)}$	$\check{(<0.05)}$	n.a	$\check{(<0.05)}$
	Cha.	1	3101	1	0	2.9	$\check{(<0.05)}$	$\check{(<0.05)}$	n.a	$\check{(<0.05)}$
	Hun.	4.022	12472	3	3.4	0.1	$\check{(<0.05)}$	$\check{(<0.05)}$	n.a	$\check{(<0.05)}$
	Chur.	16.954	52573	6	49.8	0	$\mathbf{X}(0.09)$	$\check{(<0.05)}$	n.a	$\check{(<0.05)}$
T4	Dup.	0.029	216	0	0.2	62.6	$\mathbf{X}(0.45)$	$\mathbf{X}(0.19)$	$\check{(<0.05)}$	n.a
	Tim.	52.76	391586	4	182.2	47.4	$\check{(<0.05)}$	$\check{(<0.05)}$	$\check{(<0.05)}$	n.a
	Com.	8.313	61701	5	10.2	68.3	$\check{(<0.05)}$	$\check{(<0.05)}$	$\check{(<0.05)}$	n.a
	Reo.	0.077	570	0	0.3	64.6	$\check{(<0.05)}$	$\mathbf{X}(0.59)$	$\check{(<0.05)}$	n.a
	Fil.	5.633	41805	3	14	79.9	$\check{(<0.05)}$	$\check{(<0.05)}$	$\check{(<0.05)}$	n.a
	Sev.	3.835	28466	4	1	56	$\check{(<0.05)}$	$\check{(<0.05)}$	$\check{(<0.05)}$	n.a
	Cha.	12.861	95455	4	52.2	89.7	$\check{(<0.05)}$	$\check{(<0.05)}$	$\check{(<0.05)}$	n.a
	Hun.	2305.868	17114149	30	58094.7	96	$\check{(<0.05)}$	$\check{(<0.05)}$	$\check{(<0.05)}$	n.a
	Chur.	27249.773	202247816	204	320023.5	91.9	$\check{(<0.05)}$	$\check{(<0.05)}$	$\check{(<0.05)}$	n.a

Table 13: Netbeans Ecosystem Complexity Metrics Comparison and Mann-whitney test results.

μ :mean,

\sum :sum, \hat{x} :median, σ :standard deviation, %:percentage

Types	Metric	μ	\sum	\hat{x}	σ	%	T1	T2	T3	T4
T1	Dup.	0.086	67	0	0.4	2.5	n.a	$\mathbf{X}(0.39)$	$\mathbf{X}(0.24)$	$\mathbf{X}(0.86)$
	Tim.	92.759	71981	10	219.1	2.3	n.a	$\checkmark(<0.05)$	$\mathbf{X}(0.15)$	$\checkmark(<0.05)$
	Com.	4.687	3637	3	4.1	2.4	n.a	$\checkmark(<0.05)$	$\mathbf{X}(0.83)$	$\checkmark(<0.05)$
	Reo.	0.054	42	0	0.3	1.9	n.a	$\mathbf{X}(0.1)$	$\mathbf{X}(0.58)$	$\checkmark(<0.05)$
	Fil.	1.735	1346	1	13.2	0.8	n.a	$\checkmark(<0.05)$	$\checkmark(<0.05)$	$\checkmark(<0.05)$
	Sev.	4.314	3348	3	1.5	3.1	n.a	$\mathbf{X}(0.66)$	$\checkmark(<0.05)$	$\checkmark(<0.05)$
	Cha.	1.085	842	1	0.4	2	n.a	$\mathbf{X}(0.99)$	$\mathbf{X}(0.26)$	$\checkmark(<0.05)$
	Hun.	4.405	3418	3	7	0.5	n.a	$\checkmark(<0.05)$	$\mathbf{X}(0.13)$	$\checkmark(<0.05)$
	Chur.	5.089	3949	2	12.5	0.3	n.a	$\checkmark(<0.05)$	$\checkmark(<0.05)$	$\checkmark(<0.05)$
T2	Dup.	0.067	16	0	0.3	0.6	$\mathbf{X}(0.39)$	n.a	$\mathbf{X}(0.73)$	$\mathbf{X}(0.39)$
	Tim.	111.9	26856	16	308.6	0.9	$\checkmark(<0.05)$	n.a	$\checkmark(<0.05)$	$\mathbf{X}(0.41)$
	Com.	4.433	1064	3	4	0.7	$\checkmark(<0.05)$	n.a	$\checkmark(<0.05)$	$\checkmark(<0.05)$
	Reo.	0.079	19	0	0.3	0.9	$\mathbf{X}(0.1)$	n.a	$\mathbf{X}(0.11)$	$\mathbf{X}(0.97)$
	Fil.	8.804	2113	2	42.7	1.3	$\checkmark(<0.05)$	n.a	$\checkmark(<0.05)$	$\checkmark(<0.05)$
	Sev.	4.362	1047	3	1.5	1	$\mathbf{X}(0.66)$	n.a	$\checkmark(<0.05)$	$\checkmark(<0.05)$
	Cha.	1.075	258	1	0.3	0.6	$\mathbf{X}(0.99)$	n.a	$\mathbf{X}(0.5)$	$\checkmark(<0.05)$
	Hun.	21.887	5253	8	62.7	0.7	$\checkmark(<0.05)$	n.a	$\checkmark(<0.05)$	$\checkmark(<0.05)$
	Chur.	32.263	7743	8	125.8	0.7	$\checkmark(<0.05)$	n.a	$\checkmark(<0.05)$	$\checkmark(<0.05)$
T3	Dup.	0.074	620	0	0.4	23.3	$\mathbf{X}(0.24)$	$\mathbf{X}(0.73)$	n.a	$\checkmark(<0.05)$
	Tim.	87.033	728642	9	233.6	23.8	$\mathbf{X}(0.15)$	$\checkmark(<0.05)$	n.a	$\checkmark(<0.05)$
	Com.	4.73	39599	3	4.3	26.5	$\mathbf{X}(0.83)$	$\checkmark(<0.05)$	n.a	$\checkmark(<0.05)$
	Reo.	0.06	499	0	0.3	22.7	$\mathbf{X}(0.58)$	$\mathbf{X}(0.11)$	n.a	$\checkmark(<0.05)$
	Fil.	1.306	10932	1	5.1	6.8	$\checkmark(<0.05)$	$\checkmark(<0.05)$	n.a	$\checkmark(<0.05)$
	Sev.	4.021	33666	3	1.4	31.4	$\checkmark(<0.05)$	$\checkmark(<0.05)$	n.a	$\checkmark(<0.05)$
	Cha.	1.065	8917	1	0.3	21	$\mathbf{X}(0.26)$	$\mathbf{X}(0.5)$	n.a	$\checkmark(<0.05)$
	Hun.	5.15	43115	3	12.4	5.8	$\mathbf{X}(0.13)$	$\checkmark(<0.05)$	n.a	$\checkmark(<0.05)$
	Chur.	6.727	56317	2	22	4.9	$\checkmark(<0.05)$	$\checkmark(<0.05)$	n.a	$\checkmark(<0.05)$
T4	Dup.	0.113	1959	0	0.7	73.6	$\mathbf{X}(0.86)$	$\mathbf{X}(0.39)$	$\checkmark(<0.05)$	n.a
	Tim.	128.833	2237319	13	332.8	73	$\checkmark(<0.05)$	$\mathbf{X}(0.41)$	$\checkmark(<0.05)$	n.a
	Com.	6.058	105202	4	6.7	70.4	$\checkmark(<0.05)$	$\checkmark(<0.05)$	$\checkmark(<0.05)$	n.a
	Reo.	0.094	1639	0	0.4	74.5	$\checkmark(<0.05)$	$\mathbf{X}(0.97)$	$\checkmark(<0.05)$	n.a
	Fil.	8.408	146019	4	25.1	91	$\checkmark(<0.05)$	$\checkmark(<0.05)$	$\checkmark(<0.05)$	n.a
	Sev.	3.982	69159	3	1.4	64.5	$\checkmark(<0.05)$	$\checkmark(<0.05)$	$\checkmark(<0.05)$	n.a
	Cha.	1.871	32494	2	1.2	76.4	$\checkmark(<0.05)$	$\checkmark(<0.05)$	$\checkmark(<0.05)$	n.a
	Hun.	40.195	698022	13	98.3	93.1	$\checkmark(<0.05)$	$\checkmark(<0.05)$	$\checkmark(<0.05)$	n.a
	Chur.	61.893	1074830	15	178.6	94	$\checkmark(<0.05)$	$\checkmark(<0.05)$	$\checkmark(<0.05)$	n.a

Table 14: Apache and Netbeans Ecosystems Complexity Metrics Comparison and Mann-whitney test results.

μ :mean, \sum :sum, \hat{x} :median, σ :standard deviation, %:percentage

Types	Metric	μ	\sum	\hat{x}	σ	%	T1	T2	T3	T4
T1	Dup.	0.043	118	0	0.3	3.9	n.a	X (0.09)	X (0.16)	\checkmark (<0.05)
	Tim.	91.909	252198	6	250.6	6.5	n.a	\checkmark (<0.05)	\checkmark (<0.05)	\checkmark (<0.05)
	Com.	4.449	12208	3	4.5	5.1	n.a	\checkmark (<0.05)	\checkmark (<0.05)	\checkmark (<0.05)
	Reo.	0.06	164	0	0.3	5.3	n.a	X (0.07)	\checkmark (<0.05)	\checkmark (<0.05)
	Fil.	1.201	3296	1	7	1.5	n.a	\checkmark (<0.05)	\checkmark (<0.05)	\checkmark (<0.05)
	Sev.	3.675	10085	4	1.4	6.4	n.a	X (0.97)	X (0.17)	\checkmark (<0.05)
	Cha.	1.024	2810	1	0.2	1.9	n.a	\checkmark (<0.05)	\checkmark (<0.05)	\checkmark (<0.05)
	Hun.	3.981	10924	3	4.3	0.1	n.a	\checkmark (<0.05)	\checkmark (<0.05)	\checkmark (<0.05)
	Chur.	14.894	40870	5	42.2	0	n.a	\checkmark (<0.05)	\checkmark (<0.05)	\checkmark (<0.05)
T2	Dup.	0.03	44	0	0.2	1.5	X (0.09)	n.a	\checkmark (<0.05)	\checkmark (<0.05)
	Tim.	114.632	170573	9	296.4	4.4	\checkmark (<0.05)	n.a	\checkmark (<0.05)	X (0.15)
	Com.	4.943	7355	3	4.6	3.1	\checkmark (<0.05)	n.a	X (0.72)	\checkmark (<0.05)
	Reo.	0.073	108	0	0.3	3.5	X (0.07)	n.a	\checkmark (<0.05)	X (0.47)
	Fil.	5.095	7581	2	25.4	3.6	\checkmark (<0.05)	n.a	\checkmark (<0.05)	\checkmark (<0.05)
	Sev.	3.637	5412	4	1.3	3.4	X (0.97)	n.a	X (0.44)	X (0.1)
	Cha.	4.099	6100	2	18.7	4.1	\checkmark (<0.05)	n.a	\checkmark (<0.05)	\checkmark (<0.05)
	Hun.	474.881	706623	12	12481.7	3.8	\checkmark (<0.05)	n.a	\checkmark (<0.05)	\checkmark (<0.05)
	Chur.	11902.19	17710459	62	366988	8	\checkmark (<0.05)	n.a	\checkmark (<0.05)	\checkmark (<0.05)
T3	Dup.	0.058	670	0	0.4	22.3	X (0.16)	\checkmark (<0.05)	n.a	\checkmark (<0.05)
	Tim.	73.21	839942	6	215.8	21.6	\checkmark (<0.05)	\checkmark (<0.05)	n.a	\checkmark (<0.05)
	Com.	4.647	53311	3	4.3	22.2	\checkmark (<0.05)	X (0.72)	n.a	\checkmark (<0.05)
	Reo.	0.052	600	0	0.3	19.5	\checkmark (<0.05)	\checkmark (<0.05)	n.a	\checkmark (<0.05)
	Fil.	1.221	14013	1	4.4	6.6	\checkmark (<0.05)	\checkmark (<0.05)	n.a	\checkmark (<0.05)
	Sev.	3.919	44966	3	1.4	28.4	X (0.17)	X (0.44)	n.a	\checkmark (<0.05)
	Cha.	1.048	12018	1	0.3	8.1	\checkmark (<0.05)	\checkmark (<0.05)	n.a	\checkmark (<0.05)
	Hun.	4.845	55587	3	10.7	0.3	\checkmark (<0.05)	\checkmark (<0.05)	n.a	\checkmark (<0.05)
	Chur.	9.491	108890	3	32.3	0	\checkmark (<0.05)	\checkmark (<0.05)	n.a	\checkmark (<0.05)
T4	Dup.	0.088	2175	0	0.6	72.3	\checkmark (<0.05)	\checkmark (<0.05)	\checkmark (<0.05)	n.a
	Tim.	106.056	2628905	9	297.9	67.6	\checkmark (<0.05)	X (0.15)	\checkmark (<0.05)	n.a
	Com.	6.733	166903	4	8	69.6	\checkmark (<0.05)	\checkmark (<0.05)	\checkmark (<0.05)	n.a
	Reo.	0.089	2209	0	0.4	71.7	\checkmark (<0.05)	X (0.47)	\checkmark (<0.05)	n.a
	Fil.	7.577	187824	3	22.4	88.3	\checkmark (<0.05)	\checkmark (<0.05)	\checkmark (<0.05)	n.a
	Sev.	3.938	97625	3	1.3	61.8	\checkmark (<0.05)	X (0.1)	\checkmark (<0.05)	n.a
	Cha.	5.162	127949	2	29	85.9	\checkmark (<0.05)	\checkmark (<0.05)	\checkmark (<0.05)	n.a
	Hun.	718.58	17812171	16	31804.5	95.8	\checkmark (<0.05)	\checkmark (<0.05)	\checkmark (<0.05)	n.a
	Chur.	8202.463	203322646	28	175548.3	91.9	\checkmark (<0.05)	\checkmark (<0.05)	\checkmark (<0.05)	n.a

Nyholm 2007) and a well-known measure of bug impact.

Overall, the complexity of bug types in terms of the number of duplicates is as follows: $T4_{dup}^1 \gg T1_{dup}^3 > T3_{dup}^2 \gg T2_{dup}^4$.

Fixing time

The fixing time metric represents the time it took for the bug report to go from the *new* state to the *closed* state. If the bug report is reopened, then the time it took for the bug to go from the *assigned* state to the *closed* state is added to the first time. A bug report can be reopened several times and all the times are added. In this section, the time is expressed in days (Weiss et al. 2007; Zhang et al. 2012; Zhang, Gong, and Versteeg 2013).

When combined, both ecosystem amounts in the following order $T2_{time}^4 > T4_{time}^1 \gg T1_{time}^3 \gg T3_{time}^2$. These findings contradict the finding of Saha *et al.*, however, they did not study the Netbeans ecosystem in their paper (Saha, Khurshid, and Perry 2014).

Comments

The “number of comments” metric refers to the comments that have been posted by the community on the project tracking system. This third process metric evaluates the complexity of a given bug in a sense that if it takes more comments (explanation) from the reporter or the assignee to provide a fix, then the bug must be more complex to understand. The number of comments has been shown to be useful in assessing the complexity of bugs (Zhang, Gong, and Versteeg 2013; Zhang et al. 2012). It is also used in bug prediction approaches (D’Ambros, Lanza, and Robbes 2010; Bhattacharya and Neamtiu 2011).

When combining both ecosystems, the results are: $T4_{comment}^1 \gg T2_{comment}^4 > T3_{comment}^2 \gg T1_{comment}^3$.

Bug Reopening

The bug is reopening metric counts how many times a given bug gets reopened. If a bug report is reopened, it means that the fix was arguably hard to come up with or the report was hard to understand (Zimmermann et al. 2012; Shihab et al. 2010; Lo 2013).

When combined, however, the order does change: $T4_{reop}^1 > T2_{reop}^4 > T1_{reop}^3 \gg T3_{reop}^2$.

Severity

The severity metric reports the degree of impact of the report on the software. Predicting the severity of a given report is an active research field (Menzies and Marcus 2008; Guo2010; Lamkanfi et al. 2010; Tian, Lo, and Sun 2012; Valdivia Garcia and Shihab 2014; Havelund, Holzmann, and Joshi 2015) and it helps to prioritization of fixes (Xuan et al. 2012). The severity is a textual value (blocker, critical, major, normal, minor, trivial) and the Mann-Whitney test only accepts numerical input. Consequently, we had to assign numerical values to each severity. We chose to assign values from 1 to 6 for trivial, minor, normal, major, critical and blocker severities, respectively.

The bug type ordering according to the severity metrics is: $T4_{sev}^1 \gg T3_{sev}^2 \gg T2_{sev}^4 > T1_{sev}^3$, $T2_{sev}^4 > T1_{sev}^3 \gg T3_{sev}^2 \gg T4_{sev}^1$ and $T4_{sev}^1 \gg T3_{sev}^2 > T1_{sev}^3 > T2_{sev}^4$ for Apache, Netbeans, and both combined, respectively.

Files impacted

The number of files impacted measures how many files have been modified for the bug report to be closed.

Overall, T4 impacts more files than T2 while T1 and T2 impacts only 1 file ($T4_{files}^1 \gg T2_{files}^3 \gg T3_{files}^2 \leq T1_{files}^4$).

Changesets

The changeset metrics registers how many changesets (or commits/patch/fix) have been required to close the bug report. In the project tracking system, changesets to resolve the bug are proposed and analysed by the community, automated quality insurance tools and the quality insurance team itself. Each changeset can be either accepted and applied to the source code or dismissed. The number of changesets (or versions of a given changeset) it takes before integration can hint us about the complexity of the fix. In case the bug report gets reopen, and new changesets proposed, the new changesets (after the reopening) are added to the old ones (before the reopening).

Overall, T4 bugs are the most complex bugs regarding the number of submitted changesets ($T4_{changesets}^1 \gg T2_{changesets}^3 \gg T3_{changesets}^2 \gg T1_{changesets}^4$).

While results have been published on the bug-fix patterns (Pan, Kim, and Whitehead 2008), smell introduction (Tufano et al. 2015; Eyolfson, Tan, and Lam 2011), to the best of our knowledge, no one interested themselves in how many iterations of a patch was required to close a bug report beside us.

Hunks

The hunks metric counts the number of consecutive code blocks of modified, added or deleted lines in textual files. Hunks are used to determine, in each file, how many different places a developer has modified. This metric is widely used for bug insertion prediction (Kim, Zimmermann, Pan, et al. 2006a; Jung, Oh, and Yi 2009; Rosen, Grawi, and Shihab 2015) and bug-fix comprehension (Pan, Kim, and Whitehead 2008). In our ecosystems, there is a relationship between the number of files modified and the hunks. The number of code blocks modified is likely to rise as to the number of modified files as the hunks metric will be at least 1 per file.

We found that T2 and T4 bugs, that requires many files to get fixed, are the ones that have significantly higher scores for the hunks metric; Apache ecosystem: $T4_{hunks}^1 \gg T2_{hunks}^2 \gg T3_{hunks}^3 \gg T1_{hunks}^4$, Netbeans ecosystem: $T4_{hunks}^1 \gg T2_{hunks}^3 \gg T3_{hunks}^2 \gg T1_{hunks}^4$, and overall $T4_{hunks}^1 \gg T2_{hunks}^2 \gg T1_{hunks}^4 \gg T3_{hunks}^3$.

Churns

The last metric, churns, counts the number of lines modified. The churn value for a line change should be at least two as the line has to be deleted first and then added back with the modifications. Once again, this is a widely used metric in the field (Kim, Zimmermann, Pan, et al. 2006a; Pan, Kim, and Whitehead 2008; Jung, Oh, and Yi 2009; Rosen, Grawi, and Shihab 2015).

Once again, T4 and T2 are the ones with the most churns; Apache ecosystem $T4_{churns}^1 \gg T2_{churns}^2 \gg T1_{churns}^4 > T3_{churns}^3$, Netbeans ecosystem: $T4_{churns}^1 \gg T2_{churns}^3 \gg T3_{churns}^2 \gg T1_{churns}^4$ and overall : $T4_{churns}^1 \gg T2_{churns}^2 \gg T1_{churns}^4 \gg T3_{churns}^3$.

To determine which type is the most complex, we counted how many times each bug type obtained each position in our nine rankings and multiply them by 4 for the

first place, 3 for the second, 2 for the third and 1 for the fourth place.

We did the same simple analysis of the rank of each type for each metric, to take into account the frequency of bug types in our calculation, and multiply both values. The complexity scores we calculated are as follows: 1330, 1750, 2580 and 7120 for T1, T2, T3 and T4 bugs, respectively.

Considering that type 4 bugs are (a) the most common, (b) the most complex and (c) not a type we intuitively know about; we decided to kick start our research into the different type of bugs and their impact by predicting whether an incoming bug report type 4 or not.

9.3 Empirical Validation

In this section, we present the results of our experiences and interpret them to answer our two research questions.

9.3.1 Are T4 bug predictable at submission time?

To answer this question, we used as features the words in the bug description contained in a bug report. We removed the stopwords (i.e. the, or, she, he) and truncated the remaining words to their roots (i.e. writing becomes write, failure becomes fail and so on). We experimented with 1-gram, 2-gram, and 3-gram words weighted using tf-idf. To build the classifier, we examined three machine learning techniques that have shown to yield satisfactory results in related studies: SVM, Random forest and linear regression (Wei, Zimmermann, and Zeller 2007; Alencar, Abebe, and McIntosh 2014; Nam, Pan, and Kim 2013).

To answer **RQ₁**, we analyse the accuracy of predictors aiming at determining the type of a bug at submission time (i.e. when someone submits the bug report).

Tables 15, 16 and 17 presents the results obtained while building classifiers for the most complex type of bug. According to the complexity analysis conducted in section 9.2.2, the most complex type of bug, in terms of duplicate, time to fix, comments, reopening, files changed, severity, changesets, churns, and hunks is T4.

To answer our research question, we built nine different classifiers using three different machine learning techniques: Linear regression, support vector machines and random forest for ten different projects (5 from each ecosystem).

We selected the top 5 projects of each ecosystem with regard to their bug report count (Ambari, Cassandra, Flume, HBase and Hive for Apache; Cnd, Editor, Java, JavaEE and Platform for Netbeans). For each machine learning techniques, we built classifiers using the text contained in the bug report and the comment of the first 48 hours as they are likely to provide additional insights on the bug itself. We eliminate the stop-words of the text and trim the words to their semantical roots using wordnet. We experimented with 1-gram, 2-gram, and 3-gram words, weighted using tf/idf.

The feature vectors are fed to the different machine learning techniques in order to build a classifier. The data is separated into two parts with a 60%-40% ratio. The 60% part is used for training purposes while the 40% is used for testing purposes. During the training process we use the ten-folds technique iteratively and, for each iteration, we change the parameters used by the classifier building process (cost, mtry, etc). At the end of the iterations, we select the best classifier and exercise it against the second part of 40%. The results we report in this section are the performances of the nine classifiers trained on 60% of the data and classifying the remaining 40%. The performances of each classifier are examined in terms of true positive, true negative, false negative and false positive classifications. True positives and negative numbers refer to the cases where the classifier correctly classify a report. The false negative represents the number of reports that are classified as non-T4 while they are and false positive represents the number of reports classified as T4 while they are not. These numbers allow us to derive three common metrics: precision, recall and f₁ measure.

$$precision = \frac{TP + FN \cap TP + FP}{TP + FP} \quad (12)$$

$$recall = \frac{TP + FN \cap TP + FP}{TP + FN} \quad (13)$$

$$f_1 = \frac{2TP}{2TP + FP + FN} \quad (14)$$

The performances of each classifier are compared to a tenth classifier. This last classifier is a random classifier that randomly predicts the type of a bug. As we are in a two classes system (T4 and non-T4), 50% of the reports are classified as T4 by the random classifier. The performances of the random classifier itself are presented in table 18.

Finally, we compute the Cohen’s Kappa metric (Fleiss and Cohen 1973) for each classifier. The Kappa metric compares the observed accuracy and the expected accuracy to provide a less misleading assessment of the classifier performance than precision alone.

$$kappa = \frac{(observedaccuracy - expectedaccuracy)}{1 - expectedaccuracy} \quad (15)$$

The observed accuracy represents the number of items that were correctly classified, according to the ground truth, by our classifier. The expected accuracy represents the accuracy obtained by a random classifier.

For the first three classifiers (SVM, linear regression and random forest with a 1-gram grouping of stemmed words) the best classifier the random forest one with 77.63% F_1 measure. It is followed by SVM (77.19%) and, finally, linear regression (76.31%). Regardless of the technique used to classify the report, there is no significant difference between ecosystems. Indeed, the p-values obtained with chi-square tests are above 0.05, and a p-value below 0.05 is a marker of statistical significance. While random forest emerges as the most accurate classifier, the difference between the three classifiers is not significant (p-value = 0.99).

For the second three classifiers (SVM, linear regression and random forest with 2-grams grouping of stemmed words) the best classifier is once again random forest with 77.34% F_1 measure. It is followed by SVM (76.91%) and, finally, linear regression (76.25%). As for the first three classifiers, the difference between the classifiers and the ecosystems are not significant. Moreover, the difference in performances between 1 and 2 grams are not significant either.

Finally, the last three classifiers (SVM, linear regression and random forest with 3-grams grouping of stemmed words) the best classifier is once again random forest with 77.12% F1-measure. It is followed by SVM (76.72%) and, finally, linear regression (75.89%). Again, the difference between the classifiers and the ecosystems are not significant. Neither are the differences in results between 1, 2 and 3 grams.

Each one of our nine classifiers improves upon the random one on all projects and by a large margin ranging from 20.73% to 22.48% regarding F-Measure.

The last measure of performance for our classifier is the computation of the Cohen’s Kappa metric presented in table 19.

The table presents the results of the Cohen’s kappa metric for each of our nine

Table 15: Support Vector Machine, Linear Regression and Random Forest based classifiers performances while using 1 gram.
TP: True positive, TN: True Negative, FN: False Negative, FP: False Positive

Project	Reports	T4 Reports	TP	TN	FN	FP	Precision	Recall	F1
Support Vector Machine									
Ambari	829	540	539	4	1	285	65.41%	99.81%	79.03%
Cassandra	340	199	193	5	6	136	58.66%	96.98%	73.11%
Flume	133	80	79	9	1	44	64.23%	98.75%	77.83%
HBase	357	215	213	4	2	138	60.68%	99.07%	75.27%
Hive	272	191	191	0	0	81	70.22%	100.00%	82.51%
Cnd	1105	805	753	25	52	275	73.25%	93.54%	82.16%
Editor	666	478	455	16	23	172	72.57%	95.19%	82.35%
Java	1090	693	676	37	17	360	65.25%	97.55%	78.20%
JavaEE	585	287	258	52	29	246	51.19%	89.90%	65.23%
Platform	969	573	467	110	106	286	62.02%	81.50%	70.44%
Total	6346	4061	3824	262	237	2023	65.40%	94.16%	77.19%
Linear Regression									
Ambari	829	540	514	14	26	275	65.15%	95.19%	77.35%
Cassandra	340	199	194	5	5	136	58.79%	97.49%	73.35%
Flume	133	80	60	17	20	36	62.50%	75.00%	68.18%
HBase	357	215	212	5	3	137	60.74%	98.60%	75.18%
Hive	272	191	103	40	88	41	71.53%	53.93%	61.49%
Cnd	1105	805	762	26	43	274	73.55%	94.66%	82.78%
Editor	666	478	459	16	19	172	72.74%	96.03%	82.78%
Java	1090	693	683	13	10	384	64.01%	98.56%	77.61%
JavaEE	575	287	271	30	16	258	51.23%	94.43%	66.42%
Platform	969	573	486	102	87	294	62.31%	84.82%	71.84%
Total	6336	4061	3744	268	317	2007	65.10%	92.19%	76.31%
Random Forest									
Ambari	829	540	514	13	26	276	65.06%	95.19%	77.29%
Cassandra	337	199	191	12	8	126	60.25%	95.98%	74.03%
Flume	133	80	76	8	4	45	62.81%	95.00%	75.62%
HBase	357	215	212	9	3	133	61.45%	98.60%	75.71%
Hive	272	191	190	3	1	78	70.90%	99.48%	82.79%
Cnd	1105	805	803	4	2	296	73.07%	99.75%	84.35%
Editor	666	478	476	3	2	185	72.01%	99.58%	83.58%
Java	1090	693	682	26	11	371	64.77%	98.41%	78.12%
JavaEE	575	287	252	59	35	229	52.39%	87.80%	65.63%
Platform	969	573	437	153	136	242	64.36%	76.27%	69.81%
Total	6333	4061	3833	291	228	1981	65.93%	94.39%	77.63%

Table 16: Support Vector Machine, Linear Regression and Random Forest based classifiers performances while using 2 grams.
TP: True positive, TN: True Negative, FN: False Negative, FP: False Positive

Project	Reports	T4 Reports	TP	TN	FN	FP	Precision	Recall	F1
Support Vector Machine									
Ambari	829	540	525	12	15	277	65.46%	97.22%	78.24%
Cassandra	323	199	189	11	10	113	62.58%	94.97%	75.45%
Flume	133	80	74	15	6	38	66.07%	92.50%	77.08%
HBase	357	215	205	23	10	119	63.27%	95.35%	76.07%
Hive	272	191	171	15	20	66	72.15%	89.53%	79.91%
Cnd	1105	805	731	34	74	266	73.32%	90.81%	81.13%
Editor	666	478	455	30	23	158	74.23%	95.19%	83.41%
Java	1090	693	664	58	29	339	66.20%	95.82%	78.30%
JavaEE	575	287	238	69	49	219	52.08%	82.93%	63.98%
Platform	969	573	461	110	112	286	61.71%	80.45%	69.85%
Total	6319	4061	3713	377	348	1881	66.37%	91.43%	76.91%
Linear Regression									
Ambari	829	540	510	19	30	270	65.38%	94.44%	77.27%
Cassandra	340	199	140	55	59	86	61.95%	70.35%	65.88%
Flume	142	89	59	23	30	30	66.29%	66.29%	66.29%
HBase	357	215	90	100	125	42	68.18%	41.86%	51.87%
Hive	272	191	176	8	15	73	70.68%	92.15%	80.00%
Cnd	1105	805	745	26	60	274	73.11%	92.55%	81.69%
Editor	666	478	453	27	25	161	73.78%	94.77%	82.97%
Java	1090	693	606	106	87	291	67.56%	87.45%	76.23%
JavaEE	575	287	245	70	42	218	52.92%	85.37%	65.33%
Platform	815	573	449	121	124	121	78.77%	78.36%	78.57%
Total	6191	4070	3473	555	597	1566	68.92%	85.33%	76.25%
Random Forest									
Ambari	829	540	511	20	29	269	65.51%	94.63%	77.42%
Cassandra	340	199	176	22	23	119	59.66%	88.44%	71.26%
Flume	133	80	72	21	8	32	69.23%	90.00%	78.26%
HBase	351	215	208	12	7	124	62.65%	96.74%	76.05%
Hive	272	191	190	0	1	81	70.11%	99.48%	82.25%
Cnd	1105	805	794	9	11	291	73.18%	98.63%	84.02%
Editor	666	478	471	6	7	182	72.13%	98.54%	83.29%
Java	1099	702	673	43	29	354	65.53%	95.87%	77.85%
JavaEE	575	287	238	84	49	202	54.09%	82.93%	65.47%
Platform	1002	606	444	163	162	233	65.58%	73.27%	69.21%
Total	6372	4103	3777	382	326	1887	66.68%	92.05%	77.34%

Table 17: Support Vector Machine, Linear Regression and Random Forest based classifiers performances while using 3 grams.
TP: True positive, TN: True Negative, FN: False Negative, FP: False Positive

Project	Reports	T4 Reports	TP	TN	FN	FP	Precision	Recall	F1
Support Vector Machine									
Ambari	829	540	520	15	20	274	65.49%	96.30%	77.96%
Cassandra	340	199	193	11	6	130	59.75%	96.98%	73.95%
Flume	133	80	74	8	6	45	62.18%	92.50%	74.37%
HBase	357	215	208	24	7	118	63.80%	96.74%	76.89%
Hive	272	191	175	14	16	67	72.31%	91.62%	80.83%
Cnd	1105	805	725	34	80	266	73.16%	90.06%	80.73%
Editor	666	478	454	22	24	166	73.23%	94.98%	82.70%
Java	1090	693	662	61	31	336	66.33%	95.53%	78.30%
JavaEE	575	287	256	45	31	243	51.30%	89.20%	65.14%
Platform	969	573	461	111	112	285	61.80%	80.45%	69.90%
Total	6336	4061	3728	345	333	1930	65.89%	91.80%	76.72%
Linear Regression									
Ambari	829	540	505	26	35	263	65.76%	93.52%	77.22%
Cassandra	340	199	176	21	23	120	59.46%	88.44%	71.11%
Flume	133	80	68	18	12	35	66.02%	85.00%	74.32%
HBase	357	215	91	99	124	43	67.91%	42.33%	52.15%
Hive	272	191	185	5	6	76	70.88%	96.86%	81.86%
Cnd	1105	805	747	22	58	278	72.88%	92.80%	81.64%
Editor	666	478	448	31	30	157	74.05%	93.72%	82.73%
Java	1090	693	667	55	26	342	66.11%	96.25%	78.38%
JavaEE	575	287	256	51	31	237	51.93%	89.20%	65.64%
Platform	969	573	468	102	105	294	61.42%	81.68%	70.11%
Total	6336	4061	3611	430	450	1845	66.18%	88.92%	75.89%
Random Forest									
Ambari	829	540	500	22	40	267	65.19%	92.59%	76.51%
Cassandra	340	199	188	14	11	127	59.68%	94.47%	73.15%
Flume	133	80	70	23	10	30	70.00%	87.50%	77.78%
HBase	357	215	206	24	9	118	63.58%	95.81%	76.44%
Hive	272	191	189	1	2	80	70.26%	98.95%	82.17%
Cnd	1105	805	755	27	50	273	73.44%	93.79%	82.38%
Editor	666	478	453	32	25	156	74.38%	94.77%	83.35%
Java	1090	693	665	77	28	320	67.51%	95.96%	79.26%
JavaEE	575	287	241	135	46	215	52.85%	83.97%	64.87%
Platform	969	573	443	132	130	264	62.66%	77.31%	69.22%
Total	6336	4061	3710	425	351	1850	66.73%	91.36%	77.12%

Table 18: Random classifier.

True positive, TN: True Negative, FN: False Negative, FP: False Positive

Project	Reports	T4 Reports	TP	TN	FN	FP	Precision	Recall	F1
Ambari	828	540	249	158	291	131	65.53%	46.11%	54.13%
Cassandra	339	199	111	68	88	73	60.33%	55.78%	57.96%
Flume	132	80	32	31	48	22	59.26%	40.00%	47.76%
HBase	356	215	105	68	110	74	58.66%	48.84%	53.30%
Hive	271	191	85	40	106	41	67.46%	44.50%	53.63%
Cnd	1104	805	393	159	412	141	73.60%	48.82%	58.70%
Editor	665	478	230	94	248	94	70.99%	48.12%	57.36%
Java	1089	693	365	205	328	192	65.53%	52.67%	58.40%
JavaEE	574	287	122	148	165	140	46.56%	42.51%	44.44%
Platform	968	573	277	194	296	202	57.83%	48.34%	52.66%
Total	6335	4061	1969	1165	2092	1110	63.95%	48.49%	55.15%

Table 19: Cohen's Kappa for each classifier

Type	Gram	TP T4	TN T4	Observed Accuracy	Expected Accuracy	Kappa	Interpretation
SVM	1	3824	262	0.64	0.49	0.30	Fair
	2	3713	377	0.64	0.49	0.30	Fair
	3	3728	345	0.64	0.49	0.29	Fair
Linear Regression	1	3744	268	0.63	0.49	0.27	Fair
	2	3473	555	0.63	0.49	0.28	Fair
	3	3611	430	0.64	0.49	0.28	Fair
Random Forest	1	3833	291	0.65	0.49	0.31	Fair
	2	3777	382	0.66	0.49	0.32	Fair
	3	3710	425	0.65	0.49	0.31	Fair

classifiers. The metric is computed using the observed accuracy and the expected accuracy. The observed accuracy, in our bi-class system (i.e. T4 or not), is the number of correctly classified type 4 added to the number of correctly classified non-T4 bugs over the total of reports. The expected accuracy follows the same principle but using the classification from the random classifier. The expected accuracy is constant as the random classifier predicts 50% of the reports as T4 and 50% as non-T4. Finally, the obtained Cohen’s Kappa measures range from 0.27 to 0.32. While there is no unified way to interpret the result of the Cohen’s kappa statistic, Landis and Koch considers 0-0.20 as slight, 0.21-0.40 as fair, 0.41-0.60 as moderate, 0.61-0.80 as substantial, and 0.81-1 as almost perfect (Landis and Koch 1977). Consequently, all of our classifiers show a fair improvement over a random classification regarding accuracy and a major improvement regarding F1-measure.

9.3.2 What are the best predictors of type 4 bugs?

In this section, we answer our second research question: *What are the best predictors of type 4 bugs*. To do so, we extracted the best predictor of type 4 bugs for each one of the extracted grams (1, 2 and 3) for each of our ten test projects (Five Apache, Five Netbeans). Then, we manually investigated the source code and the reports of these ten software projects to determine why a given word is a good predictor of type 4 bug. In the remaining of this section, we present our findings by project and then provide a conclusion on the best predictors of type 4 bugs.

Ambari

Ambari is aimed at making Hadoop management simpler by developing software for provisioning, managing, and monitoring Apache Hadoop clusters. One of the most acclaimed features of Ambari is the ability to visualise clusters’ health, according to user-defined metric, with heat maps. These heat maps give a quick overview of the system.

Figure 50 shows a screenshot of such a heat map.

At every tested gram (i.e. 1, 2 and 3) the word “heat map” is a strong predictor of type 4 bugs. The heat map feature is a complex feature as it heavily relies on the underlying instrumentation of Hadoop and the consumption of many log format

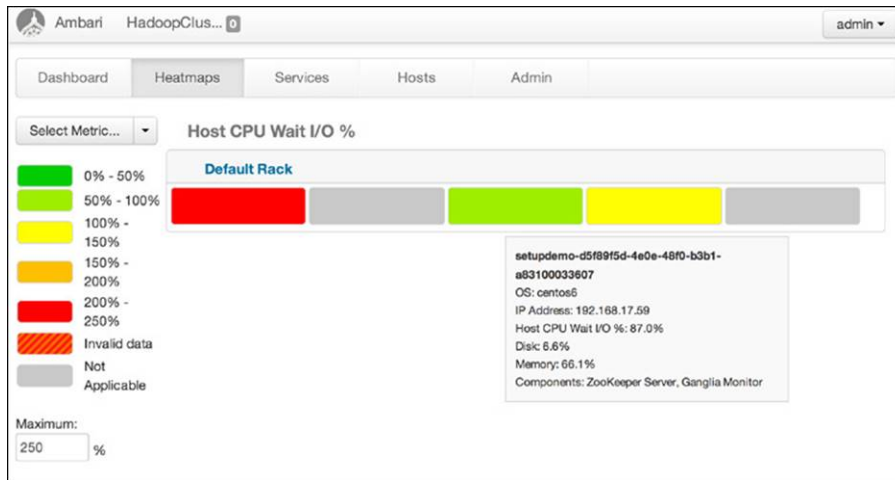


Figure 50: Ambari heatmap

too, for example, extracts the remaining free space on a disk or the current load on a CPU.

Another word that is a strong predictor of type 4 bug is “nagio”. Nagio is a log monitoring server belonging to the Apache constellation. It is used as an optional add-on for Ambari and, as for the heat map, is very susceptible to log format change and API breakage.

Versions of the “nagio” and “heatmap” keywords include: “heatmap displai”, “ambari heatmap”, “fix nagio”, “nagio test”, “ambari heatmap displai”, “fix nagio test”.

Cassandra

Cassandra is a database with high scalability and high availability without compromising performance. While extracting the unique word combinations from the report of Cassandra, one word which is a strong predictor of type 4 bug is “snapshot”.

As described in the documentation, in Cassandra terms, *a snapshot first flushes all in-memory writes to disk, then makes a hard link of the SSTable files for each keyspace. You must have enough free disk space on the node to accommodate making snapshots of your data files. A single snapshot requires little disk space. However, snapshots can cause your disk usage to grow more quickly over time because a snapshot prevents old obsolete data files from being deleted. After the snapshot is complete, you can move the backup files to another location if needed, or you can leave them in place.*

The definition gives the reader an insight into how complex this feature used regarding integration with the host system and how coupled it is to the Cassandra, data model.

Other versions of the “snapshot” keyword include “snapshot sequenti”, “make snapshot”, “snapshot sequenti repair”, “make snapshot sequenti”.

Flume

Flume is a distributed, reliable, and available service for efficiently collecting, aggregating, and moving large amounts of log data.

One word which is a good predictor of type 4 in flume is “upgrad” and the 2-grams (upgrad flume) and the 3-grams (“upgrad flume to”) versions. Once again for the Apache dataset, a change in the software that induce a change in the underlying data model or data store, which is often the case when you upgrade flume to a new version, is a good indicator of the report complexity and the impact of said report on the sourcecode in terms of number of locations fixed.

On the reports manually analysed, Flume’s developers and users have a hard time upgrading to new versions in a sense that logs and dashboard get corrupted or disappear post-upgrade. Significant efforts are then made to prevent such losses in the subsequent version.

HBase

HBase is a Hadoop database, a distributed, scalable, big data store provided by Apache. The best predictor of type 4 bug in HBase is “bloom” as in “bloom filters”. Bloom filters are a probabilistic data structure that is used to test whether an element is a member of a set (Broder and Mitzenmacher 2004). Such a feature is hard to implement and hard to test because of its probabilistic nature. Much feature commits (i.e. commit intended to add a feature) and fix commits (i.e. commit intended to fix a bug) belonging to the HBase source code are related to the bloom filters. Given the nature of the feature, it is not surprising to find the word “bloom” and its 2-, 3-grams counterparts (“on Bloom”, “Bloom filter”, “on Bloom filter”) as a good predictor of type 4 bug.

Hive

Hive is a data warehouse software facilitates reading, writing, and managing large datasets residing in distributed storage using SQL. Hive is different from its Apache counterpart as the words that are the best predictors of type 4 bugs do not translate into a particular feature of the product but are directly the name of the incriminated part of the system: thrift. Thrift is a software framework, for scalable cross-language services development, combines a software stack with a code generation engine to build services that work efficiently and seamlessly between C++, Java, Python, PHP, Ruby, Erlang, Perl, Haskell, C#, Cocoa, JavaScript, Node.js, Smalltalk, OCaml and Delphi and other languages. While Thrift is supposed to solve many compatibility issues when building clients for a product such as Hive, it is the cause of many major problems in Hive. The top predictors for type 4 bugs in Hive are “thrifthttpcliservic” and “thriftbinarycliservic”.

As Hive, and its client, are built on top of Thrift it makes sense that issues propagating from Thrift induce major refactoring and fixed across the whole Hive source code.

Cnd

The CND projects is a part of the Netbeans IDE and provide support for C/C++. The top two predictors of type 4 bugs are (1) parallelism and (2) observability of c/c++ code. In each gram, we can find reference to the parallel code being problematic while developed and executed via the Netbeans IDE: “parallel comput”, “parallel”, “parallel comput advis”. The other word, related to the observability of c/c++ code inside the Netbeans IDE is “Gizmo”. “Gizmo” is the codename for the C/C++ Observability Tool built on top of D-Light Toolkit. We can find occurrences of “Gizmo” in each gram: “gizmo” and “gizmo monitor” for example.

Once again, a complex cross-concern feature with a high impact on the end-user (i.e., the ability to code, execute and debug parallel code inside Netbeans) is the cause of most of the type 4 bugs and mention of said feature in the report is a bug predictor of types of the bug.

Editor

The Editor component of Netbeans is the component which is handling all the textual edition, regardless of the programming language, in Netbeans. For this component, the type 4 bugs are most likely related to the “trailing white spaces” and “spellcheck” features.

While these features do not, at first sight, be as complex as, for example, parallelism debugging, they have been the cause of the majority of type 4 bugs. Upon manual inspection of the related code² in the Editor component of Netbeans the complexity of these feature becomes evident. Indeed, theses features behave differently for almost each type of text-file and textboxes inside Netbeans. For example, the end-user expects the spellchecking feature of the IDE to kick in while typing a comment inside a code file but not on the code itself. A similar example can be described for the identification and removing of trailing white spaces where users wish the trailing white spaces to be deleted in c/c++ code but not, for example, while typing HTML or a commit message.

Each new language supported or add-on supported by the Netbeans IDE and leveraging the features of the Editor component is susceptible to be the cause of a major refactoring to have a coherent behaviour regarding “trailing white spaces” and “spell checking”.

Java

The Java component of Netbeans is responsible for the Java support of Netbeans in the same fashion as CND is responsible for c/c++ support. For this particular component, the set of features that are a good predictor of type 4 are the ones related to the Java autocompletion and navigation optimisation. The autocompletion has to be able to provide suggestions in a near-instantaneous manner if it is to be useful to the developer. To provide near-instantaneous suggestion on modest machines and despite the depth of the Java API, Netbeans developers opted of a statistical autocompletion. The autocompletion *remembers* which of its suggestions you used before and only provide the ones you are the most likely to want to be based on your previous usage. Also, each suggestion is companioned with a percentage which describes the number of time you pick a given a suggestion over the other. One can envision a such a system

²<https://netbeans.org/projects/editor/>

can be tricky to implement on new API being added in the Java language at each upgrade. Indeed, when a new API comes to light following a Java upgrade on the developer's machine, then, the autocompletion has to make these new API appears in the autocompletion despite their 0% chosen rate. The 0% being linked to the fact that this suggestion was not available thus far and not to the fact that the developer never picked it. When the new suggestion, related to the new API, has been ignored a given number of time, then, it can be safely removed from the list of suggestions.

Implementation of optimisations related to autocompletion and navigations are the root causes of many type 4 bugs, and we can find them in the gram extracted words that are good predictor: “implement optim”, “move otim”, “optim import implement”, “call hierarchy implement”.

JavaEE

The JavaEE component of Netbeans is responsible for the support of the JavaEE in Netbeans. This module is different from the CND and JAVA module in a sense that it uses and expands many functionalities from the JAVA component. For the JavaEE component, the best predictor of type 4 bugs is the hibernate and webscoket features which can be found in many gram forms: “hibern revers”, “websocket endpoint”, “hibern”, “websocket”, “implement hibern revers”, “hibern revers engin”.

Hibernate is an ORM that enables developers to write applications whose data outlives the application process more easily. As an Object/Relational Mapping (ORM) framework, Hibernate is concerned with data persistence as it applies to relational databases (via JDBC).

The shortcoming of Netbeans leading to most of the type 4 bugs is related to the annotation based persistence of Hibernate where developers can annotate their class attributes with the name of the column they wish the value of the attribute to be persisted. While the annotation mechanism is supported by Java, it is not possible *compile* annotation and makes sure that their statically sound. Consequently, much tooling around annotation has to be developed and maintained accordingly to new databases updates. Such tooling, for example, is responsible for querying the database model to make sure that the annotated columns exists and can store the attribute data type-wise.

Platform

The last netbeans component we analyzed is the one named Platform. *The NetBeans Platform is a generic framework for Swing applications. It provides the “plumbing” that, before, every developer had to write themselves—saving state, connecting actions to menu items, toolbar items and keyboard shortcuts; window management, and so on.* (<https://netbeans.org/features/platform/>)

The best predictor of type 4 bug in the platform component is the “filesystem” word which refers to the ability of any application built atop of Platform to use the filesystem for saves and such.

What we can conclude for this second research question is that the best predictor of type 4 bugs is the mention of a cross-concern, complex, widely used feature in the targeted system. Reports mentioning said feature are likely to create a type 4 structure with many bugs being fixed in the same set of files. One noteworthy observation is that the 2- and 3-grams extraction do not add much to the precision about the 1-gram extraction as seen the first research question. Upon the manual analysis required for this research question, we can deduct why. Indeed, the problematic features of a given system are identified with a single word (i.e. hibernate, filesystem, spellcheck, ...). While the 2- and 3-grams classifiers do not provide an additional performance in the classification process, they still become handy when trying to target which part of the feature a good predictor of type 4 (“implement optim”, “gizmo monitor”, “heatmap displai”, ...).

9.4 Threats to Validity

The selection of target systems is one of the common threats to validity for approaches that perform qualitative and quantitative analyses.

While is it possible the selected programs share common properties that we are not aware of and therefore, invalidate our results, this is highly unlikely. Indeed, our dataset is composed of 388 open source systems.

In addition, we see a threat to validity that stems from the fact that we only used open-source systems. The results may not be generalizable to industrial systems. We intend to undertake these studies in future work.

9.5 Chapter Summary

In this chapter, we proposed a taxonomy of bugs and performed an empirical study on two large open source datasets: the Netbeans IDE and the Apache Software Foundation's projects. Our study aimed to analyse: (1) the proportion of each type of bugs; (2) the complexity of each type in terms of severity, reopening and duplication; and (3) the required time to fix a bug depending on its type. The key findings are that Type 4 account for 61% of the bugs and have a bigger impact on software maintenance tasks than the other three types.

In the next chapter, we start a discussion covering several topics ranging from challenges surrounding the adoption of tools to our advice for university-industry research collaboration.

Chapter 10

Discussion

10.1 Adoption of Fault Prevention Tools in Industry

In the introduction of this thesis, we state that the large adoption of tools related to automated maintenance by practitioners remains limited.

The ultimate goal is to develop techniques and tools to help software developers detect, correct, and prevent bugs effectively and efficiently. Despite the recent advances in the field, the literature shows that many existing software maintenance tools have yet to be adopted by industry (Lewis et al. 2013; Foss and Murphy 2015; Layman, Williams, and Amant 2007; Ayewah et al. 2007; Ayewah and Pugh 2008; Johnson et al. 2013; Norman 2013; Hovemeyer and Pugh 2004; Lopez and Hoek 2011). We believe that the following factors cause this:

- Integration with the developer’s workflow: Most existing maintenance tools ((Kim, Pan, and Whitehead 2006; Ayewah et al. 2008; Findbugs 2015; Moha et al. 2010; Palma, Nayrolles, and Moha 2013; Nayrolles 2013; Nayrolles et al., n.d.; Nayrolles, Moha, and Valtchev 2013, Mathieu Nayrolles, Beaudry, et al. (2015)) are some noticeable examples) are not integrated well with the work flow of software developers (i.e., coding, testing, debugging, committing). Using these tools, developers have to download, install and understand them to achieve a given task. They would constantly need to switch from one workspace to another for different tasks (i.e., feature location with a command line tool,

development and testing code with an IDE, development and testing front end code with another IDE and a browser, etc.) (Robertson et al. 2004; Robertson, Lawrance, and Burnett 2006; Beckwith et al. 2006).

- **Corrective actions:** The outcome of these tools does not always lead to corrective actions that the developers can implement. Most of these tools return several results that are often difficult to interpret by developers. Take for example, FindBugs (Hovemeyer and Pugh 2004), a popular bug detection tool. This tool detects hundreds of bug signatures and reports them using an abbreviated code such as *CO_COMPARETO_INCORRECT_FLOATING*. Using this code, developers can browse the FindBug’s dictionary and find the corresponding definition *This method compares double or float values using pattern like this: $val1 > val2 \sim 1 : val1 < val2 ? -1 : 0$* . While the detection of this bug pattern is accurate, the tool does not propose any corrective actions to the developers that can help them fix the problem. Moreover, it has been reported in the literature that the output of existing maintenance tools tends to be verbose at the point where developers decide to simply ignore them (Arai et al. 2014; Kim and Ernst 2007b; Kim and Ernst 2007a; Ayewah and Pugh 2010; Shen, Fang, and Zhao 2011).
- **Leverage of historical data:** These tools do not leverage a large body of knowledge that already exists in open source systems. For defect prevention, for example, the state of the art approaches consists of adapting statistical models built for one project to another project (Lo 2013; Nam, Pan, and Kim 2013). As argued by Lewis et al. (Lewis et al. 2013) and Johnson et al. (Johnson et al. 2013), approaches based solely on statistical models are perceived by developers as black box solutions. Developers are less likely to trust the output of these tools.

We believe that the tools and approaches proposed in this thesis tackle most of the factors that hinder the adoption of software maintenance tools by practitioners.

10.2 The Right-Time For Just-In-Time Fault Prevention

Software quality insurance encompasses different activities (e.g., defect detection and prediction, clone detection, source code inspection, unit testing, implementing new requirements, etc.) that play an important role in producing high quality software. It exists several classes of maintenance (adaptive, perfective, corrective, preventive) which, when combined, represent up to 70% of the overall cost of any given project (Health, Social and Research 2002). The international organization for standardization (ISO) defines software maintenance to be the modification of a software product after delivery to correct faults, to improve performances or other attributes, or to adapt the product to a changed environment (ISO/IEC-14764:2006 2006). Researchers, however, have proposed approaches and tools to support practitioners in doing these actions before the delivery. Developers can leverage these approaches to, for examples, detect anti-patterns, clones, defects or, performance bottlenecks before shipping a new version of their software. More specifically, developers have access to tools and IDE-plugins that supports the detection of anti-patterns, clones and defects patterns. Quality assurance team have access to approaches that generate lists of packages or files that are likely to contain defects.

In recent years, several research areas (mining bug repositories, bug analysis, patch analysis, bug reproduction) related to software maintenance have been exposed to a novel idea: “Just-In-Time Software Quality Insurance.” “Just-In-Time Quality Assurance” is the concept that software maintenance (or quality insurance) cannot be a separate task of developing software but interleaved with day-to-day programming (Kamei et al. 2013; Yang et al. 2015; Tourani and Adams 2016). Indeed, software project and repositories not only became large, but they are also now ultra-large (Northrop et al. 2006). Thinking about software quality insurance activities as a dedicated chunk of times where the code is reviewed, risky package analyzed, anti-patterns refactored and code clones removed is impractical because the mental models and thoughts processes that led to these changes or design decisions are long gone. Consequently, “Just-In-Time Software Maintenance” aims to perform these activities as soon as a change is made, so the decisions motivating the changes are still fresh in the mind of practitioners.

In this paper, we first propose a short historic of the just-in-time concepts. Then, we present a comparison of the different moments where just-in-time maintenance can be applied. We do so to characterize what would be the right time, in terms of productivity and resulting quality, to apply just-in-time software maintenance approaches.

10.2.1 Just-In-Time: From Toyota Plants to Software Quality

The term Just-In-Time (JIT) have first been associated with manufacturing in the 1970s. More specifically, Just-In-Time manufacturing was first developed and perfected within the Toyota manufacturing plants by Taiichi Ohno as a means of meeting consumer demands with minimum delays and waste (i.e., time and resources) (Suzaki 1987). To do so, JIT manufacturing relies on several management philosophies and tools: continuous improvement (product-oriented layout of plants, division of systems, simplicity), elimination of waste (overproduction, waiting time, inventory waste, transportation, product defects), Hausukipingu (clean workspaces), Kabans (pulling the right number of items from the right shelves at just the right time), Jidoka (autonomous machines with judgment capabilities) and Andons (signal problems for corrective action).¹

Some of these ideas have long been transposed to software engineering. For example, continuous improvement is one of the cornerstones of the agile manifesto (Fowler and Highsmith 2001), integrated development environment (IDE) are Jidokas in a sense that they auto-complete us and auto-correct us based on past behavior and, unit tests, bug report management systems and quality assurance (QA) bots are Andons. Hausukipingu and Kabans are more difficultly transposable to software manufacturing. The last principle, elimination of waste, is one that is the more closely related to JIT software maintenance. Indeed, in JIT software maintenance and JIT quality assurance, researchers want to reduce the number of defects, clones, and bad design while they are still fresh in the mind of developers. Consequently, product defects are lowered and time is saved.

¹We kept the Japanese version of most of the words as they are used without translation in the literature.

10.2.2 Types of Just-In-Time Quality Insurance

In this section, we describe three different moments where JIT software maintenance could take place: real-time, commit-time and integration time. For each, we will divide our analyze on four parts: description, influential papers and, advantages and pitfalls.

Real-Time

Real-Time software quality insurance tools operate directly inside developers' IDE using IDE plugins. The rationale behind IDE-plugin and real-time software quality insurance is that it allows warning developers of potentially hazardous code as they write it and, consequently, save time and strengthen the overall quality of the software.

The adoption of such tools is, however, limited in the industry.

Johnson *et al* found that static analysis tools to find bugs produce too many warnings, do not provide corrective actions and are perceived as black-boxes by users (Johnson et al. 2013). Take, for example, FindBugs (Hovemeyer and Pugh 2004), a popular bug detection plugin. This plugin detects hundreds of bug signatures and reports them using an abbreviated code such as `CO_COMPARETO_INCORRECT_FLOATING` \hookrightarrow . Using this code, developers can browse the FindBug's dictionary and find the corresponding definition “*This method compares double or float values using pattern like this: $val1 > val2 ? 1 : val1 < val2 ? -1 : 0$* ”. While the detection of this bug pattern is accurate, the tool does not propose any corrective actions to the developers that can help them fix the problem. Their findings have since been confirmed many times (Arai et al. 2014; Ayewah and Pugh 2010; Shen, Fang, and Zhao 2011).

In the particular case of clone detection, Latoza *et al.* (Latoza, Venolia, and DeLine 2006) found that there exist six different reasons that trigger the use of clones (e.g., copy and paste of code examples, a reimplementations of the same functionality in a different language, etc.). Developers are aware that they are creating clones in five out of six situations. In such cases, warnings provided by IDE-based local detection techniques can be quite disturbing (Ko et al. 2006). Finally, using IDE-plugins can lead to context and workspace switching which can hinder the productivity of developers (Robertson, Lawrance, and Burnett 2006; Beckwith et al. 2006).

An overwhelming majority of approaches target the Eclipse IDE despite the fact that eclipse is only the fifth most used development environment. Indeed, in

2016 the StackOverflow developers survey² found that usage of development environment among 46,613 responses were as follows: Notepad++³ (35.6%), Visual Studio⁴ (35.6%), Sublime Text⁵ (31.0%), Vim⁶ (26.1%) and Eclipse⁷ (22.7%). Developers were allowed to choose multiple answers, hence the over 100% total. The average developer, still according to the StackOverflow survey, uses between two and three development environments. The second most investigated IDE when it comes to building real-time quality insurance plugins is Netbeans⁸. Netbeans ranks 11th with 8.1%.

Arguably, Eclipse and Netbeans are targeted because they are open-source and easy to develop for. An interesting point, however, is the fact that among the top-5 development environments, three (Notepad++, Sublime Text and, Vim) are text editors. Text-editors, at the opposite of IDE, treats code as text with only few added features like coloration or the ability to call an external build suite. The declining use of specialized IDE is correlated to the rise of a particular pedigree of developer: Full Stack Developers. Full Stack Developers, as their name suggests, manipulate each layer of their software solutions stack. Consequently, they can found themselves editing files written in a data description language for their databases, a backend language such as Php, Ruby or Java, a front-end language such as Typescript and diverse configuration files in JSON or XML. Instead of mastering as many specialized IDE as they use languages, full stack developers turned to the least common denominator and use one text editor that can read and edit any types of text files.

It is, in our opinion, fascinating that researchers focus on tools, approaches and techniques that could fit into developers' IDE without considering that (a) developers are generally reluctant to the use of such tools and, (b) the majority of developers do not use IDEs.

²<http://stackoverflow.com/research/developer-survey-2016#technology-development-environments>

³<https://notepad-plus-plus.org/>

⁴<https://www.visualstudio.com/>

⁵<https://www.sublimetext.com/>

⁶<http://www.vim.org/>

⁷<https://eclipse.org/>

⁸<https://netbeans.org/>

Integration-Time

Another time where just-in-time quality insurance have been applied is at integration-time. Integration-time refers to the times where new code modifications reach a central repository. Performing just-in-time quality insurance at integration-time is the path of least resistance when it comes to change developer processes with the dual aim of saving time and improving software quality. Indeed, integration-time approaches monitor the central code repository and perform a quality evaluation based on code or process metrics. When the quality evaluation is complete, a report is emitted, generally by email. Consequently, developers do not have to change their processes because they do not install any tools, plugins and still use text-editors. Another advantage of integration-time just-in-time quality insurance approaches is that do not execute themselves on developers' workstation. Indeed, they can monitor repositories from a remote server with adequate specifications in terms of computational power. Real-Time approaches are bound to the performances of developers' workstations.

Commit Guru is a popular example of integration-time just-in-time quality insurance approach (Rosen, Grawi, and Shihab 2015). Commit guru monitors Github repositories and builds statistical models based on code metrics to determine, for each project, thresholds above which a code modification is likely to introduce a defect into the code. Also, many commercial tools propose integration-time just-in-time quality insurance. Codeclimate,⁹ Codacy,¹⁰ Scrutinizer¹¹ and Coveralls¹² are some examples. These tools will perform various tasks such as executing unit test suites, computing quality metrics, performing clone detection and, provide a report by email.

In our opinion, the problem with integration-time just-in-time quality insurance approach is that the detection occurs too late in the development process. Once the code reaches the central repository, they can be pulled by other members of the development team, further complicating the removal and management of potential defects, clones or anti-patterns. Moreover, the asynchronous way in which integration-time just-in-time software quality maintenance operates can lead to the same context-switching and workspace switching as real-time approaches (Robertson, Lawrance, and Burnett 2006; Beckwith et al. 2006). Indeed, email reports will be asynchronously

⁹<https://codeclimate.com/>

¹⁰<https://codacy.com/>

¹¹<https://scrutinizer-ci.com/>

¹²<https://coveralls.io/>

received by developers. Consequently, it is likely that developers would have started a new task and will have to build back the thoughts processes that led risky design decisions.

Commit-Time

Commit-Time just-in-time software quality insurance is the last time at which just-in-time software quality insurance can operate. Each time a developer makes a commit, the changes are intercepted using a pre-commit hook. Pre-commit hooks are custom scripts set to fire off when certain important actions of the versioning process occur. There are two groups of hooks: client-side and server-side. Client-side hooks are triggered by operations such as committing and merging, whereas server-side hooks run on network operations such as receiving pushed commits. These hooks can be used for all sorts of reasons such as checking compliance with coding rules or automatic run of unit test suites. The pre-commit hook runs before the developer specifies a commit message. It is used to inspect the modifications that are about to be committed. To the best of our knowledge, we are the first to conceptualize commit-time for just-in-time software quality insurance.

By design, pre-commit hooks treat code modification as text and kicks in seamlessly during the versioning process. Consequently, every approach that currently takes place at real- or integration-time could be migrated at commit-time without significant effort. Also, commit-time just-in-time software quality insurance would provide recommendations interactively during the versioning operations. Providing recommendations during versioning would not interfere with developers thoughts processes as commits are bite-sized units of work that are potentially ready to be shared with the rest of the organization (O’Sullivan and Bryan 2009). Commits usually mark the end of a given task or subtask as the developer is ready to version the source code. Thus, undesirable side-effects of real-time software quality insurance would be avoided while retaining its benefits. Also, commit-Time just-in-time software quality insurance intervenes before the code reaches the central repository and became pullable by other members of the organization. Finally, the quality analysis happens synchronously and prevent developers to begin new tasks. Their design decisions would still be fresh in their mind presented with the results of the quality analysis at commit-time.

10.2.3 Conclusion

In this section, we presented three different times at which just-in-time software quality insurance can be implemented: real-time, integration-time and commit-time. While it exists real-time and integration-time approaches and tool, to the best of our knowledge, we are the first to conceptualize what a commit-time approach would look like.

Real-time approaches interrupt the developers with many warnings and recommendations. Moreover, they assume that developers use IDE and the majority do not. Integration-time approaches allow developers to keep their processes as they seamlessly monitor the central repository and perform asynchronous quality analysis when new changes are received. When developers receive the report of integration-time approaches, they are likely to have started new development tasks and will have to build back the thoughts processes that led risky design decisions.

Commit-Time just-in-time software quality insurance approaches would be an efficient trade-off between real- and integration-time as they would address major factors that contribute to the slow adoption of quality assurance tools. Indeed, they would fit in the day-to-day workflow of developers (i.e. coding, testing, debugging, committing) and will not hinder their productivity by yielding context or workspace switches. In addition, commit-time approaches do not rely on IDE but the versioning system. Following the rationale that full stack developers turned to text editor that can read and edit any types of text files, commit-time approaches would rely on another text-based technology that any current development team uses: code versioning. Finally, developers are used to having an interactive versioning process as, for example, they can be notified that they do not have the latest version of the source-code locally in the case that another member of the organization committed modifications. In such a case, developers would have to retrieve the modifications and merge them to their own before being allowed to share their work with the organization. We that providing quality insurance recommendations following the same process would not impact developers significantly while achieving the quality objectives of just-in-time quality insurance. Commit-time *is* the right time for just-in-time software quality insurance.

10.3 University-Industry Research Collaboration

Software maintenance tasks continue to be challenging and costly, which explains the growing interest in industry-scale techniques for the detection and prevention of software defects. This is valid for most software organizations including Ubisoft, one of the world's leading video game development companies. The company specializes in the design and implementation of high-budget video games such as Prince of Persia, Far Cry and Assassin's Creed, and is heavily invested in software development and maintenance tasks. To continue its expansion with more and bigger games, while preserving quality, Ubisoft embarked on a research project in collaboration with Concordia University to explore software quality control techniques that can detect or even prevent the insertion of bugs, preferably, before system modifications reach the central software repository, i.e., at commit-time.

The project should achieve a number of requirements. The techniques must operate at commit-time, embedded within Ubisoft's code versioning systems. This requirement ensures that the accompanying tool fits well with the developers' workflow, eliminating the need to download and install external tools, which typically requires extensive settings and a high learning curve. The new tool should also be scalable to work with Ubisoft complex ecosystem, constituted of software systems that are highly coupled containing millions of files and commits, developed and maintained by more than 8,000 developers scattered across 29 locations in six continents. With these requirements in mind, the research team at Concordia started working with Ubisoft software developers to understand better the industrial context and the type of systems that are used. Together, we defined the scope of the project and formed a team that includes students and developers from Ubisoft. The lead student was assigned to work on Ubisoft premises on a full-time basis during the project. The project lasted approximately eight months. Regular meetings were held to follow progress and make the necessary adjustments.

The lessons learned through this industrial collaboration are numerous. In this talk, we discuss the most important ones that are summarized in what follows:

10.3.1 Deep understanding of the project requirements

Throughout the design of CLEVER, it was important to have a close collaboration between the research team and the Ubisoft developers. This allowed the research team to understand well the requirements of the project. Through this collaboration, both the research and development teams quickly realized that existing work in the literature was not sufficient to address the project's requirements. In addition, existing studies were mainly tested using open source systems, which may be quite different in structure and size from large industrial systems. In our case, we found that a deep understanding of Ubisoft ecosystem was an important enabler for many decisions we made in this project including the fact that CLEVER operates on multiple systems and that it uses a two-phase mechanism. It was also important to come up with a solution that integrates well with the workflow of Ubisoft developers. This required the development of CLEVER in a way it integrates well with the entire suite of Ubisoft's version control systems. The key lesson here is to understand well the requirements of a project and its complexity.

10.3.2 Understanding the benefits of the project to both parties

Understanding how the project benefits the company and the university helps both parties align their vision and work towards a common goal and set of objectives. From Ubisoft's perspective, the project provides sound mechanisms for building reliable systems. In addition, the time saved from detecting and fixing defects can be shifted to the development of new functionalities that add value to Ubisoft customers. For the university research team, the project provides an excellent opportunity for gaining a better understanding of the complexity of industrial systems and how research can provide effective and practical solutions. Also, working closely with software developers helps uncover the practical challenges they face within the company's context. Companies vary significantly in terms of culture, development processes, maturity levels, etc. Research effort should be directed to develop solutions that overcome these challenges, while taking into account the organizational context.

10.3.3 Focusing in the Beginning on Low-Hanging Fruits

Low-hanging fruits are quick fixes and solutions. We found that it is a good idea to showcase some quick wins early in the project to show the potential of the proposed solutions. At the beginning of the project, we applied the two-phase process of CLEVER to some small systems with a reasonable number of commits. We showed that the approach improved over the use of metrics alone. We also showed that CLEVER was able to make suggestions on how to fix the detected risky commits. This encouraged us to continue on this path and explore additional features. We continued to follow an iterative and incremental process throughout the project where knowledge transition between the University and Ubisoft teams is done on a regular basis. Building a Strong Technical Team: Working on industrial projects requires all sort of technical skills including programming in various programming languages, the use of tools, tool integration, etc. The strong technical skills of the lead student of this project were instrumental in the success of this project. It should be noted that Ubisoft systems are programmed using different languages, which complicated the code matching phase of CLEVER. In addition, Ubisoft uses multiple bug management and version control systems. Downloading, processing, and manipulating commits from various environment requires excellent technical abilities.

10.3.4 Communicating effectively

During the development of CLEVER, we needed to constantly communicate the steps of our research to developers and project owners. Adopting a communication strategy suitable to each stakeholder was important. For example, in our meetings with management, we focused more on the ability of CLEVER to improve code quality and reduce maintenance costs instead of the technical details of the proposed approach. Developers, on the other hand, were interested in the potential of CLEVER and its integration with their work environment.

In this talk, we share our experience conducting a research project at Ubisoft. The project consists of developing techniques and a tool for detecting defects before they reach the code repository. Our approach, called CLEVER, achieves this in two phases using a combination of metric-based machine learning models and clone detection. CLEVER is being deployed at Ubisoft.

10.3.5 Managing change

Any new initiative brings with it important changes to the way people work. Managing these changes from the beginning of the project increases the chances for tool adoption. To achieve this, we used a communication strategy that involved all the stakeholders including software developers and management to make sure that potential changes that CLEVER would bring are thoroughly and smoothly implemented, and that the benefits of change are long-lasting.

Chapter 11

Conclusion and Future Work

In this chapter, we present the conclusion of this thesis and future work that remain to be done.

Software maintenance activities such as debugging and feature enhancement are known to be challenging and costly (Pressman 2005). Studies have shown that the cost of software maintenance can reach up to 70% of the overall cost of the software development life cycle (Health, Social and Research 2002). Much of this is attributable to several factors including the increase in software complexity, the lack of traceability between the various artifacts of the software development process, the lack of proper documentation, and the unavailability of the original developers of the systems.

In this thesis, we presented three approaches that perform software maintenance at commit-time (PRECINCT, BIANCA and, CLEVER). We also presented JCHARMING that can reproduce on field crash when commit-time approaches did not catch the defect before its release. Finally, we also propose a taxonomy of bugs that could be used by researchers to categorize the research in many areas related to software maintenance.

11.1 Summary of the Findings

- We created BUMPER, an aggregated, searchable bug-fix repository that contains 1,930 projects, 732,406 resolved/fixed, 1,645,252 changesets from Eclipse, Gnome, Netbeans and the Apache Software foundation.
- We proposed PRECINCT, an incremental, online clone-detection approach that

operates at commit-time. It was able to achieve an average 97.7% precision and 100% recall while requiring a fraction of the time thanks to its incremental approach.

- We presented BIANCA, that detects risky commits and proposes potential fixes using clone-detection and dependency clustering. It was able to detect risky-commit with average of 90.75% precision and 37.15% recall at commit-time. In addition, 78% of the proposed fixes were automatically classified as qualitative using a similarity threshold with the actual fix.
- Out of the 15,316 commits BIANCA classified as *risky*, only 1,320 (8.6%) were because they were matching a defect-commit inside the same project. This supports the idea that within the same project, developers are not likely to introduce the same defect twice. Over similar projects, however, similar bugs are introduced.
- We manually reviewed 250 fixes proposed by BIANCA. We were able to identify the statements from the proposed fixes that can be reused to create fixes similar to the ones that developers had proposed in 84% of the cases.
- While the recall of BIANCA 37.15%, it is important to note that we do not claim that of issues in open-source systems are caused by project dependencies.
- We built upon BIANCA with CLEVER that combines clone- and metric-based detection of risky commit and proposes potential fixes. It significantly reduced to scalability concerns of BIANCA while obtaining an average of 79.10% precision and a 65.61% recall.
- 66% of the fixes proposed by CLEVER were accepted by software developer within Ubisoft.
- We introduced JCHARMING, an automatic bug reproduction technique that combines crash traces and directed model checking. When applied to thirty bugs from ten open source systems, JCHARMING was able to successfully reproduce 80% of the bugs. The average time to reproduce a bug was 19 minutes, which is quite reasonable, given the complexity of reproducing bugs that cause field crashes.

- Finally, we proposed a taxonomy of bugs and performed an empirical. The key findings are that Type 4 account for 61% of the bugs and have a bigger impact on software maintenance tasks than the other three types.

11.2 Future Work

11.2.1 Current Limitations

We should acknowledge that the most notable shortcoming of this thesis is the fact that we did not incorporate developers' opinions enough in our studies. Indeed, we only gathered developers opinions' in two separate occasions (BUMPER, Chapter 4 and CLEVER, Chapter 7). While their opinions were positives, we should continue to ask practitioners for their feedbacks on our works.

Another limitation of our work is that most of the approaches (BUMPER, BIANCA and, CLEVER) will most likely be ineffective if applied to a single-system. Indeed, these approaches rely on the large amount of data acquired from multiple software ecosystems to perform.

This leads to the scalability issues of our work. The model required to operate BIANCA took nearly three months using 48 Amazon Virtual Private Servers running in parallel to built and tested. While CLEVER addresses some of the scalability issues with its two-step classifier, the search of potential solution is still computationally expansive.

11.2.2 Other Possible Opportunities for Future Research

To build on this work, we additional experiment with additional (and larger) systems with the dual aim of fine-tuning the approach, and assessing the scalability of our approach when applied to even larger systems could be conducted. Also, we want to improving PRECINCT to support Type 4 clones will be a significant advantage over other clone-detectors. In addition, conducting user studies with developers to assess the concrete effectiveness of PRECINCT compared to remote and local clone detection techniques.

Conducting human studies with developers in order to gather their feedback on

BIANCA and CLEVER would be beneficial. The feedback obtained could help fine-tune the approaches. Also, examining the relationship between project cluster measures (such as betweenness) and the performance of BIANCA. Finally, another improvement to BIANCA would be to support Type 4 clones.

For BIANCA, building a feedback loop between the users and the clusters of known buggy commits and their fixes. If a fix is never used by the end-users, then we could remove it from the clusters and improve our accuracy.

For JCHARMING's, more experiments with more (and complex) bugs with the dual aim to (a) improve and fine tune the approach, and (b) assess the scalability of our approach when applied to even larger (and proprietary) systems. Finally, comparing JCHARMING to STAR (N. Chen 2013b), which is the closest approach to ours by running both methods on the same set of systems and bugs, could yield interesting results. This comparative study can provide insights into the differences in the use of symbolic analysis as it is in STAR and directed model checking, the core mechanism of JCHARMING.

Chapter 12

Appendices

12.1 Lists of the top-level open-source projects

Lists all the top-level open-source projects we analysed for our works.

12.1.1 Parsers

- Mime4j: Apache James Mime4J provides a parser, MimeStreamParser, for e-mail message streams in plain rfc822 and MIME format
- Xerces: XML parsers for c++, java and perl
- Xalan:XSLT processor for transforming XML documents into HTML, text, or other XML document types.
- FOP:Print formatter driven by XSL formatting objects (XSL-FO) and an output independent formatter.
- Droids: intelligent standalone robot framework that allows to create and extend existing droids (robots).
- Betwit: XML introspection mechanism for mapping beans to XML

12.1.2 Databases

- Drill: Schema-free SQL Query Engine for Hadoop, NoSQL and Cloud Storage
- Tez: Framework for complex directed-acyclic-graph of tasks for processing data.built atop Apache Hadoop YARN.

- HBase: Apache HBase is the Hadoop database, a distributed, scalable, big data store.
- Falcon: Falcon is a feed processing and feed management system aimed at making it easier for end consumers to onboard their feed processing and feed management on hadoop clusters.
- Cassandra: Database with high scalability and high availability without compromising performance
- Hive: Data warehouse software facilitates reading, writing, and managing large datasets residing in distributed storage using SQL
- Sqoop: Tool designed for efficiently transferring bulk data between Apache Hadoop and structured datastores such as relational databases.
- Accumulo: Sorted, distributed key/value store is a robust, scalable, high performance data storage and retrieval system.
- Lucene: Full-featured text search engine library written entirely in Java. It is a technology suitable for nearly any application that requires full-text search, especially cross-platform.
- CouchDB: Store your data with JSON documents. Access your documents and query your indexes with your web browser, via HTTP.
- Phoenix: OLTP and operational analytics in Hadoop for low latency applications
- OpenJPA: Java persistence project that can be used as a stand-alone POJO persistence layer or integrated into any Java EE
- Gora: Provides an in-memory data model and persistence for big data
- Optiq: framework that allows efficient translation of queries involving heterogeneous and federated data.
- HCatalog: Table and storage management layer for Hadoop that enables users with different data processing tools
- DdlUtils: Component for working with Database Definition (DDL) files
- Derby: Relational database implemented entirely in Java
- DBCP: Supports interaction with a relational database
- JDO: Object persistence technology

12.1.3 Web and Services

- Wicket: Server-side Java web framework
- Service Mix: The components project holds a set of JBI (Java Business Integration) components that can be installed in both the ServiceMix 3 and ServiceMix 4 containers.
- Shindig: Apache Shindig is an OpenSocial container and helps you to start hosting OpenSocial apps quickly by providing the code to render gadgets, proxy requests, and handle REST and RPC requests.
- Felix: Implement the OSGi Framework and Service platform and other interesting OSGi-related technologies under the Apache license.
- Trinidad: JSF framework including a large, enterprise quality component library.
- Axis: Web Services / SOAP / WSDL engine.
- Synapse: Lightweight and high-performance Enterprise Service Bus
- Giraph: Iterative graph processing system built for high scalability.
- Tapestry: A component-oriented framework for creating highly scalable web applications in Java.
- JSPWiki: WikiWiki engine, feature-rich and built around standard JEE components (Java, servlets, JSP).
- TomEE: Java EE 6 Web Profile certified application server extends Apache Tomcat.
- Knox: REST API Gateway for interacting with Apache Hadoop clusters.
- Flex: Framework for building expressive web and mobile applications
- Lucy: Search engine library provides full-text search for dynamic programming languages
- Camel: Define routing and mediation rules in a variety of domain-specific languages, including a Java-based Fluent API, Spring or Blueprint XML Configuration files, and a Scala DSL.
- Pivot: Builds installable Internet applications (IIAs)
- Celix: Implementation of the OSGi specification adapted to C
- Traffic Server: Fast, scalable and extensible HTTP/1.1 compliant caching proxy server.
- Apache Net: Implements the client side of many basic Internet protocols. The

purpose of the library is to provide fundamental protocol access, not higher-level abstractions.

- Sling: Innovative web framework
- Axis: Implementation of the SOAP (“Simple Object Access Protocol”) submission to W3C.
- Shale: Web application framework, fundamentally based on JavaServer Faces.
- Rave: web and social mashup engine that aggregates and serves web widgets.
- Tuscany: Simplifies the task of developing SOA solutions by providing a comprehensive infrastructure for SOA development and management that is based on Service Component Architecture (SCA) standard.
- Pluto: Implementation of the Java Portlet Specification.
- ODE: Executes business processes written following the WS-BPEL standard
- Muse: Java-based implementation of the WS-ResourceFramework (WSRF), WS-BaseNotification (WSN), and WS-DistributedManagement (WSDM) specifications.
- WS-Commons: Web Services Commons Projects
- Geronimo: Server runtime that integrates the best open source projects to create Java/OSGi
- River: Network architecture for the construction of distributed systems in the form of modular co-operating services
- Commons FileUpload: Makes it easy to add robust, high-performance, file upload capability to your servlets and web applications.
- Beehive: Java Application Framework that was designed to simplify the development of Java EE based applications.
- Aries: Java components enabling an enterprise OSGi application programming model.
- Empire Db: Relational database abstraction layer and data persistence component
- Commons Daemon: Java based daemons or services
- Click: JEE web application framework
- Stanbol: Provides a set of reusable components for semantic content management.
- CXF: Open-Source Services Framework

- Sandesha2: Axis2 module that implements the WS-ReliableMessaging specification published by IBM, Microsoft, BEA and TIBCO
- Neethi: Framework for the programmers to use WS Policy
- Rampart: Provides implementations of the WS-Sec* specifications for Apache Axis2.
- AWF: web server
- Nutch: Web crawler
- HttpClient: Designed for extension while providing robust support for the base HTTP protocol
- Portals Bridges: Portlet development using common web frameworks like Struts, JSF, PHP, Perl, Velocity and Scripts such as Groovy, JRuby, Jython, BeanShell or Rhino JavaScript.
- Stonehenge: set of example applications for Service Oriented Architecture that spans languages and platforms and demonstrates best practices and interoperability.

12.1.4 Cloud and Big data

- Whirr: Set of libraries for running cloud services
- Ambari: Aimed at making Hadoop management simpler by developing software for provisioning, managing, and monitoring Apache Hadoop clusters.
- Karaf: Karaf provides dual polymorphic container and application bootstrapping paradigms to the Enterprise.
- Hadoop: Software for reliable, scalable, distributed computing.
- Hama: framework for Big Data analytics which uses the Bulk Synchronous Parallel (BSP) computing model.
- Twill: Abstraction over Apache Hadoop YARN that reduces the complexity of developing distributed applications
- Hadoop MapReduce and Framework for easily writing applications which process vast amounts of data (multi-terabyte data-sets) in-parallel on large clusters (thousands of nodes) of commodity hardware in a reliable, fault-tolerant manner.
- Tajo: Big data relational and distributed data warehouse system for Apache Hadoop

- Sentry: System for enforcing fine grained role based authorization to data and metadata stored on a Hadoop cluster.
- Oozie: Workflow scheduler system to manage Apache Hadoop jobs.
- Solr: Provides distributed indexing, replication and load-balanced querying, automated failover and recovery, centralized configuration
- Airavata: Software framework that enables you to compose, manage, execute, and monitor large scale applications
- JClouds: Multi-cloud toolkit for the Java platform that gives you the freedom to create applications that are portable across clouds while giving you full control to use cloud-specific features.
- Impala: Native analytic database for Apache Hadoop.
- Libcloud: Python library for interacting with many of the popular cloud service providers using a unified API.
- Slider: deploy existing distributed applications on an Apache Hadoop YARN cluster
- MRUNIT: Java library that helps developers unit test Apache Hadoop map reduce jobs.
- Stratos: Framework that helps run Apache Tomcat, PHP, and MySQL applications and can be extended to support many more environments on all major cloud infrastructures
- Mesos: Abstracts CPU, memory, storage, and other compute resources away from machines
- Helix: A cluster management framework for partitioned and replicated distributed resources
- Argus: Centralized approach to security policy definition and coordinated enforcement
- DeltaCloud: API that abstracts differences between clouds
- MRQL: Query processing and optimization system for large-scale, distributed data analysis, built on top of Apache Hadoop, Hama, Spark, and Flink.
- Provisionr: create and manage pools of virtual machines on multiple clouds
- Curator: A ZooKeeper Keeper.
- ZooKeeper: Open-source server which enables highly reliable distributed coordination

- Bigtop: Infrastructure Engineers and Data Scientists looking for comprehensive packaging, testing, and configuration of the leading open source big data components.
- Yarn: split up the functionalities of resource management and job scheduling/-monitoring into separate daemons.

12.1.5 Messaging and Logging

- Activemq: Messaging queue
- Qpid: Messaging queue
- log4cxx: Logging framework for C++
- log4j: Logging framework for Java
- log4net: Logging framework for .Net
- Flume: Distributed, reliable, and available service for efficiently collecting, aggregating, and moving large amounts of log data.
- Samza: The project aims to provide a near-realtime, asynchronous computational framework for stream processing.
- Pig: Analyzing large data sets that consists of a high-level language for expressing data analysis programs.
- Chukwa: Data collection system for monitoring large distributed systems
- BookKeeper: Replicated log service which can be used to build replicated state machines.
- Apollo: Faster, more reliable, easier to maintain messaging broker built from the foundations of the original ActiveMQ.
- S4: Processes continuous unbounded streams of data.

12.1.6 Graphics

- Commons Imaging: Pure-Java Image Library
- PDFBox: Java tool for working with PDF documents.
- Batik: Java-based toolkit for applications or applets that want to use images in the Scalable Vector Graphics (SVG)
- XML Graphics Commons: consists of several reusable components used by Apache Batik and Apache FOP

- UIMA: UIMA frameworks, tools, and annotators, facilitating the analysis of unstructured content such as text, audio and video.

12.1.7 Dependency Management and build systems

- Tentacles: Downloads all the archives from a staging repo, unpack them and create a little report of what is there.
- Ivy: Transitive dependency manager
- Rat: Release audit tool, focused on licenses.
- Ant: drive processes described in build files as targets and extension points dependent upon each other
- EasyAnt: Improved integration in existing build systems
- IvyIDE: Eclipse plugin which integrates Apache Ivy's dependency management into Eclipse
- NPanday: Maven for .NET
- Maven: software project management and comprehension tool

12.1.8 Networking

- Mina:100% pure java library to support the SSH protocols on both the client and server side.
- James:Delivers a rich set of open source modules and libraries, written in Java, related to Internet mail communication which build into an advanced enterprise mail server.
- Hupa:Rich IMAP-based Webmail application written in GWT (Google Web Toolkit).
- Etch:cross-platform, language and transport-independent framework for building and consuming network services
- Commons IO: Library of utilities to assist with developing IO functionality.

12.1.9 File systems and repository

- Tika: detects and extracts metadata and text from over a thousand different file types

- OODT: Apache Object Oriented Data Technology (OODT) is a smart way to integrate and archive your processes, your data, and its metadata.
- Commons Virtual File System: Provides a single API for accessing various different file systems.
- Jackrabbit Oak: Scalable and performant hierarchical content repository
- Directory: Provides directory solutions entirely written in Java.
- SANDBOX: Subversion repository for Commons committers to function as an open workspace for sharing and collaboration.

12.1.10 Misc

- Harmony: Modular Java runtime with class libraries and associated tools.
- Mahout: Machine learning applications.
- OpenCMIS: Apache Chemistry OpenCMIS is a collection of Java libraries, frameworks and tools around the CMIS specification.
- Apache Commons: Apache project focused on all aspects of reusable Java components
- Shiro: Java security framework
- Cordova: Mobile apps with HTML, CSS & JS
- XMLBeans: Technology for accessing XML by binding it to Java types
- State Chart XML: Provides a generic state-machine based execution environment based on Harel State Tables
- excalibur: lightweight, embeddable Inversion of Control
- Commons Transaction: Transactional Java programming
- Velocity: collection of POJO
- BCEL: analyze, create, and manipulate binary Java class files
- Abdera: Functionally-complete, high-performance implementation of the IETF Atom Syndication Format
- Commons Collections: Data structures that accelerate development of most significant Java applications.
- Java Caching System: Distributed caching system written in Java
- OGNL: Object-Graph Navigation Language; it is an expression language for getting and setting properties of Java objects, plus other extras such as list projection and selection and lambda expressions.

- Anything To Triples: library that extracts structured data in RDF format from a variety of Web documents.
- Axiom: provides an XML Infoset compliant object model implementation which supports on-demand building of the object tree
- Graft: debugging and testing tool for programs written for Apache Giraph
- Hivemind: Services and configuration microkernel
- XPath: defines a simple interpreter of an expression language called XPath

Bibliography

Alencar, Daniel, Surafel Lemma Abebe, and Shane McIntosh. 2014. “An Empirical Study of Delays in the Integration of Addressed Issues.” In *Software Maintenance and Evolution (ICSME)*.

Antoniol, G., G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. 2002. “Recovering traceability links between code and documentation.” *IEEE Transactions on Software Engineering* 28 (10). IEEE Press: 970–83. doi:10.1109/TSE.2002.1041053.

Anvik, John, Lyndon Hiew, and Gail C Murphy. 2006. “Who should fix this bug?” In *Proceeding of the 28th International Conference on Software Engineering - ICSE '06*, 361. New York, New York, USA: ACM Press. doi:10.1145/1134285.1134336.

Apache Software Foundation. 2000. “Apache Struts Project.” <http://struts.apache.org/>.

———. 2014. “Apache PDFBox | A Java PDF Library.” <http://pdfbox.apache.org/>.

———. n.d. “Apache Ant.” <http://ant.apache.org/>.

Apache Software Foundation. 2011. “Hadoop.” <http://hadoop.apache.org>.

———. 2012. “Mahout: Scalable machine learning and data mining.” <http://mahout.apache.org>.

Arai, Satoshi, Kazunori Sakamoto, Hironori Washizaki, and Yoshiaki Fukazawa. 2014. “A gamified tool for motivating developers to remove warnings of bug pattern tools.” In *Proceedings - 2014 6th International Workshop on Empirical Software Engineering in Practice, IWESEP 2014*, 37–42. doi:10.1109/IWESEP.2014.17.

Artzi, Shay, Sunghun Kim, and Michael D Ernst. 2008. “Recrash: Making software failures reproducible by preserving object states.” In *Proceedings of the 22nd*

European Conference on Object-Oriented Programming, 542–65.

Avizienis, A., J.-C. Laprie, B. Randell, and C. Landwehr. 2004. “Basic concepts and taxonomy of dependable and secure computing.” *IEEE Transactions on Dependable and Secure Computing* 1 (1). IEEE: 11–33. doi:10.1109/TDSC.2004.2.

Ayewah, N., D. Hovemeyer, J.D. Morgenthaler, J. Penix, and W. Pugh. 2008. “Using Static Analysis to Find Bugs.” *IEEE Software* 25 (5): 22–29. doi:10.1109/MS.2008.130.

Ayewah, Nathaniel, and William Pugh. 2008. “A report on a survey and study of static analysis users.” In *Proceedings of the 2008 Workshop on Defects in Large Software Systems - DEFECTS '08*, 1–5. New York, New York, USA: ACM Press. doi:10.1145/1390817.1390819.

———. 2010. “The Google FindBugs fixit.” In *Proceedings of the 19th International Symposium on Software Testing and Analysis - ISTA '10*, 241. New York, New York, USA: ACM Press. doi:10.1145/1831708.1831738.

Ayewah, Nathaniel, William Pugh, J. David Morgenthaler, John Penix, and YuQian Zhou. 2007. “Evaluating static analysis defect warnings on production software.” In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering - PASTE '07*, 1–8. New York, New York, USA: ACM Press. doi:10.1145/1251535.1251536.

Bachmann, Adrian, Christian Bird, Foyzur Rahman, Premkumar Devanbu, and Abraham Bernstein. 2010. “The missing links.” In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE '10*, 97. New York, New York, USA: ACM Press. doi:10.1145/1882291.1882308.

Baier, Christel, and J-P Katoen. 2008. *Principles of Model Checking*. MIT press Cambridge.

Baker, B.S. 1995. “On finding duplication and near-duplication in large software systems.” In *Proceedings of the 2nd Working Conference on Reverse Engineering*, 86–95. IEEE Comput. Soc. Press. doi:10.1109/WCRE.1995.514697.

Balazinska, M., E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis. 1999. “Partial redesign of Java software systems based on clone analysis.” In *Proceedings of the Sixth Working Conference on Reverse Engineering*, 326–36. IEEE Comput. Soc. doi:10.1109/WCRE.1999.806971.

Basili, V.R., L.C. Briand, and W.L. Melo. 1996. “A validation of object-oriented design metrics as quality indicators.” *IEEE Transactions on Software Engineering*

22 (10). IEEE: 751–61. doi:10.1109/32.544352.

Bavota, Gabriele, Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. 2013. “The Evolution of Project Inter-dependencies in a Software Ecosystem: The Case of Apache.” In *2013 IEEE International Conference on Software Maintenance*, 280–89. IEEE. doi:10.1109/ICSM.2013.39.

Baxter, Ira D., Andrew Yahin, Leonardo Moura, Marcelo Sant’Anna, and Lorraine Bier. 1998. “Clone Detection Using Abstract Syntax Trees.” In *Proceedings of the IEEE International Conference on Software Maintenance.*, 368–77. IEEE Computer Society. <http://dl.acm.org/citation.cfm?id=850947.853341>.

Beckwith, Laura, Cory Kissinger, Margaret Burnett, Susan Wiedenbeck, Joseph Lawrance, Alan Blackwell, and Curtis Cook. 2006. “Tinkering and gender in end-user programmers’ debugging.” In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 231–40. doi:10.1145/1124772.1124808.

Bell, Jonathan, Nikhil Sarda, and Gail Kaiser. 2013. “Chronicler: Lightweight recording to reproduce field failures.” In *2013 35th International Conference on Software Engineering (ICSE)*, 362–71. IEEE. doi:10.1109/ICSE.2013.6606582.

Bellon, Stefan, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. 2007. “Comparison and Evaluation of Clone Detection Tools.” *IEEE Transactions on Software Engineering* 33 (9). IEEE: 577–91. doi:10.1109/TSE.2007.70725.

Bettenburg, Nicolas, Sascha Just, Adrian Schrter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. 2008. “What makes a good bug report?” In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 308. New York, New York, USA: ACM Press. doi:10.1145/1453101.1453146.

Bettenburg, Nicolas, Rahul Premraj, and Thomas Zimmermann. 2008. “Duplicate bug reports considered harmful ... really?” In *2008 IEEE International Conference on Software Maintenance*, 337–45. IEEE. doi:10.1109/ICSM.2008.4658082.

Bhattacharya, Pamela, and Iulian Neamtiu. 2011. “Bug-fix time prediction models: can we do better?” In *Proceeding of the International Conference on Mining Software Repositories*, 207–10. New York, New York, USA: ACM Press. doi:10.1145/1985441.1985472.

Bortis, Gerald, and Andre van der Hoek. 2013. “PorchLight: A tag-based approach to bug triaging.” In *2013 35th International Conference on Software Engineering (ICSE)*, 342–51. IEEE. doi:10.1109/ICSE.2013.6606580.

Briand, L.C., J.W. Daly, and J.K. Wust. 1999. “A unified framework for coupling

measurement in object-oriented systems.” *IEEE Transactions on Software Engineering* 25 (1). IEEE: 91–121. doi:10.1109/32.748920.

Broder, Andrei, and Michael Mitzenmacher. 2004. “Network Applications of Bloom Filters: A Survey.” *Internet Mathematics* 1 (4). A.K. Peters: 485–509. doi:10.1080/15427951.2004.10129096.

Burg, Brian, Richard Bailey, Andrew J. Ko, and Michael D. Ernst. 2013. “Interactive record/replay for web application debugging.” In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology*, 473–84. New York, New York, USA: ACM Press. doi:10.1145/2501988.2502050.

Burnstein, Ilene. 2006. *Practical Software Testing: A Process-Oriented Approach*. Springer Science & Business Media. <https://books.google.com/books?hl=en{\&}lr={\&}id=IM7iBwAAQ>

Chen, Ning. 2013a. “Star: stack trace based automatic crash reproduction.” PhD thesis, The Hong Kong University of Science; Technology.

———. 2013b. “Star: stack trace based automatic crash reproduction.” PhD thesis, Honk Kong University of Science; Technology.

Chen, Tse-hsun, Meiyappan Nagappan, Emad Shihab, and Ahmed E Hassan. 2014. “An Empirical Study of Dormant Bugs Categories and Subject Descriptors.” In *Proceedings of the International Conference on Mining Software Repository*, 82–91. doi:10.1145/2597073.2597108.

Chidamber, S.R., and C.F. Kemerer. 1994. “A metrics suite for object oriented design.” *IEEE Transactions on Software Engineering* 20 (6). IEEE: 476–93. doi:10.1109/32.295895.

Chris Vignola. 2014. “The Java Community Process(SM) Program - JSRs: Java Specification Requests - detail JSR# 352.” <https://jcp.org/en/jsr/detail?id=352>.

Clause, James, and Alessandro Orso. 2007. “A Technique for Enabling and Supporting Debugging of Field Failures.” In *Proceedings of the 29th International Conference on Software Engineering*, 261–70.

CollabNet. n.d. “Tigris.org: Open Source Software Engineering.” <http://www.tigris.org/>.

Cordy, James R. 2006. “Source transformation, analysis and generation in TXL.” In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program*

Manipulation, 1–11. New York, New York, USA: ACM Press. doi:10.1145/1111542.1111544.

Cordy, James R., and Chanchal K. Roy. 2011. “The NiCad Clone Detector.” In *Proceedings of the International Conference on Program Comprehension*, 219–20. IEEE. doi:10.1109/ICPC.2011.26.

Czerwonka, Jacek, Nachiappan Nagappan, Wolfram Schulte, and Brendan Murphy. 2013. “CODEMINE: Building a Software Development Data Analytics Platform at Microsoft.” *IEEE Software* 30 (4). IEEE: 64–71. doi:10.1109/MS.2013.68.

D’Ambros, Marco, Michele Lanza, and Romain Robbes. 2010. “An extensive comparison of bug prediction approaches.” In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, 31–41. IEEE. doi:10.1109/MSR.2010.5463279.

Dallmeier, Valentin, Andreas Zeller, and Bertrand Meyer. 2009. “Generating Fixes from Object Behavior Anomalies.” In *Proceedings of the International Conference on Automated Software Engineering*, 550–54.

Dangel, Andreas. 2000. “PMD.”

De Lucia, Andrea. 2001. “Program slicing: Methods and applications.” In *International Working Conference on Source Code Analysis and Manipulation*, 144. IEEE Computer Society.

De Moura, Leonardo, and Nikolaj Bjørner. 2008. “Z3: An efficient SMT solver.” In *Tools and Algorithms for the Construction and Analysis of Systems*, 337–40. Springer.

Dean, Thomas R., James R. Cordy, Andrew J. Malton, and Kevin A. Schneider. 2003. “Agile Parsing in TXL.” In *Proceedings of the International Conference on Automated Software Engineering*, 311–36. Kluwer Academic Publishers. doi:10.1023/A:1025801405075.

Demange, Anthony, Naouel Moha, and Guy Tremblay. 2013. “Detection of SOA Patterns.” In *Proceedings of the International Conference on Service-Oriented Computing*, 114–30.

Ducasse, Stephane, Matthias Rieger, and Serge Demeyer. 1999. “A Language Independent Approach for Detecting Duplicated Code.” In *Proceedings of the International Conference on Software Maintenance*, 109–18. <http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.31.6060>.

Dutertre, Bruno, and Leonardo De Moura. 2006. “The yices smt solver.” *Tool Paper at Http://yices.csl.sri.com/tool-Paper.pdf* 2 (2).

Dyer, Robert, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. 2013.

“Boa: a language and infrastructure for analyzing ultra-large-scale software repositories.” In *Proceedings of the International Conference on Software Engineering*, 422–31. IEEE Press. <http://dl.acm.org/citation.cfm?id=2486788.2486844>.

Edelkamp, Stefan, Stefan Leue, and Alberto Lluch-Lafuente. 2004. “Directed explicit-state model checking in the validation of communication protocols.” *International Journal on Software Tools for Technology Transfer (STTT)* 5 (2-3): 247–67. doi:10.1007/s10009-002-0104-3.

Edelkamp, Stefan, Viktor Schuppan, Dragan Bosnacki, Anton Wijs, FehnkerAnsgar, and Husain Aljazzar. 2009. *Survey on directed model checking*. Springer.

El Emam, Khaled, Walcelio Melo, and Javam C. Machado. 2001. “The prediction of faulty classes using object-oriented design metrics.” *Journal of Systems and Software* 56 (1). Elsevier Science Inc.: 63–75. doi:10.1016/S0164-1212(00)00086-8.

Eldh, Sigrid. 2001. “On Test Design.” PhD thesis, Mlardalen.

Eyolfson, Jon, Lin Tan, and Patrick Lam. 2011. “Do time of day and developer experience affect commit bugginess.” In *Proceeding of the 8th Working Conference on Mining Software Repositories - MSR '11*, 153. New York, New York, USA: ACM Press. doi:10.1145/1985441.1985464.

Findbugs. 2015. “FindBugs Bug Descriptions.” <http://findbugs.sourceforge.net/bugDescriptions.html>

Fischer, M., M. Pinzger, and H. Gall. n.d. “Populating a Release History Database from version control and bug tracking systems.” In *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.*, 23–32. IEEE Comput. Soc. doi:10.1109/ICSM.2003.1235403.

Fleiss, J. L., and J. Cohen. 1973. “The Equivalence of Weighted Kappa and the Intraclass Correlation Coefficient as Measures of Reliability.” *Educational and Psychological Measurement* 33 (3). Sage PublicationsSage CA: Thousand Oaks, CA: 613–19. doi:10.1177/001316447303300309.

Foss, Sylvie L., and Gail C. Murphy. 2015. “Do developers respond to code stability warnings?” In *Proceedings of the 25th Annual International Conference on Computer Science and Software Engineering*, 162–70. IBM Corp. <http://dl.acm.org/citation.cfm?id=2886444.2886469>.

Fowler, Martin, and Jim Highsmith. 2001. “The agile manifesto.” *Software*

Development 9 (8). [San Francisco, CA: Miller Freeman, Inc., 1993-: 28–35.

Gabel, Mark, Lingxiao Jiang, and Zhendong Su. 2008. “Scalable detection of semantic clones.” In *Proceedings of the 13th International Conference on Software Engineering*, 321–30. New York, New York, USA: ACM Press. doi:10.1145/1368088.1368132.

Girvan, M., and M. E. J. Newman. 2002. “Community structure in social and biological networks.” *Proceedings of the National Academy of Sciences* 99 (12). National Academy of Sciences: 7821–6. doi:10.1073/pnas.122653799.

Gde, Nils, and Rainer Koschke. 2009. “Incremental Clone Detection.” In *Proceedings of the 13th European Conference on Software Maintenance and Reengineering*, 219–28. IEEE. doi:10.1109/CSMR.2009.20.

Graphwalker. 2016. “GraphWalker for testers.” In *Http://graphwalker.github.io/. http://graphwalker.github.io/*.

Gyimothy, T., R. Ferenc, and I. Siket. 2005. “Empirical validation of object-oriented metrics on open source software for fault prediction.” *IEEE Transactions on Software Engineering* 31 (10). IEEE: 897–910. doi:10.1109/TSE.2005.112.

Hamill, Maggie, and Katerina Goseva-Popstojanova. 2014. “Exploring fault types, detection activities, and failure severity in an evolving safety-critical software system.” *Software Quality Journal* 23 (2): 229–65. doi:10.1007/s11219-014-9235-5.

Hassan, A.E., and R.C. Holt. 2005. “The top ten list: dynamic fault prediction.” In *Proceedings of the International Conference on Software Maintenance*, 263–72. IEEE. doi:10.1109/ICSM.2005.91.

Hassan, Ahmed E. 2009. “Predicting faults using the complexity of code changes.” In *Proceedings of the International Conference on Software Engineering*, 78–88. IEEE. doi:10.1109/ICSE.2009.5070510.

Havelund, Klaus, Gerard Holzmann, and Rajeev Joshi, eds. 2015. *NASA Formal Methods*. Vol. 9058. Lecture Notes in Computer Science. Cham: Springer International Publishing. doi:10.1007/978-3-319-17524-9.

Health, Social, and Economics Research. 2002. “The Economic Impacts of Inadequate Infrastructure for Software Testing.”

Herbold, Steffen, Jens Grabowski, Stephan Waack, and Uwe Bnting. 2011. “Improved Bug Reporting and Reproduction through Non-intrusive GUI Usage Monitoring and Automated Replaying.” In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, 232–41. IEEE.

doi:10.1109/ICSTW.2011.66.

Herraiiz, Israel, Daniel M. German, Jesus M. Gonzalez-Barahona, and Gregorio Robles. 2008. “Towards a simplification of the bug report form in eclipse.” In *Proceedings of the 2008 International Workshop on Mining Software Repositories - MSR '08*, 145. New York, New York, USA: ACM Press. doi:10.1145/1370750.1370786.

Higo, Yoshiki, Ueda Yasushi, Minoru Nishino, and Shinji Kusumoto. 2011. “Incremental Code Clone Detection: A PDG-based Approach.” In *Proceedings of the 18th Working Conference on Reverse Engineering*, 3–12. IEEE. doi:10.1109/WCRE.2011.11.

Hindle, Abram, Daniel M. German, and Ric Holt. 2008. “What do large commits tell us?: a taxonomical study of large commits.” In *Proceedings of the International Workshop on Mining Software Repositories*, 99–108. New York, New York, USA: ACM Press. doi:10.1145/1370750.1370773.

Holzmann, Gerard J. 1997. “The model checker SPIN.” *IEEE Transactions on Software Engineering* 23 (5). IEEE Computer Society: 279–95.

Hovemeyer, David. 2007. “FindBugs.”

Hovemeyer, David, and William Pugh. 2004. “Finding bugs is easy.” *ACM SIGPLAN Notices* 39 (12). ACM: 92–106. doi:10.1145/1052883.1052895.

Hummel, Benjamin, Elmar Juergens, Lars Heinemann, and Michael Conradt. 2010. “Index-based code clone detection: incremental, distributed, scalable.” In *Proceedings of the IEEE International Conference on Software Maintenance*, 1–9. IEEE. doi:10.1109/ICSM.2010.5609665.

Hunt, James W., and Thomas G. Szymanski. 1977. “A fast algorithm for computing longest common subsequences.” *Communications of the ACM* 20 (5). ACM: 350–53. doi:10.1145/359581.359603.

IBM. 2006. “T. J. Watson Libraries for Analysis (WALA).”

ISO/IEC-14764:2006. 2006. *Software Engineering – Software Life Cycle Processes – Maintenance*.

Jalbert, Nicholas, and Westley Weimer. 2008. “Automated duplicate detection for bug tracking systems.” In *2008 IEEE International Conference on Dependable Systems and Networks with FTCS and DCC (DSN)*, 52–61. IEEE. doi:10.1109/DSN.2008.4630070.

Jaygarl, Hojun, Sunghun Kim, Tao Xie, and Carl K Chang. 2010. “OCAT: Object Capture based Automated Testing.” In *Proceedings of the 19th International*

Symposium on Software Testing and Analysis, 159–70.

Jeong, Gaeul, Sunghun Kim, and Thomas Zimmermann. 2009. “Improving bug triage with bug tossing graphs.” In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 111. New York, New York, USA: ACM Press. doi:10.1145/1595696.1595715.

Jiang, Lingxiao, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. 2007. “DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones.” In *Proceedings of the 29th International Conference on Software Engineering*, 96–105. IEEE. doi:10.1109/ICSE.2007.30.

Jin, Wei, and Alessandro Orso. 2012. “BugRedux: Reproducing field failures for in-house debugging.” In *Proceedings of the 34th International Conference on Software Engineering, IEEE*, 474–84. Ieee. doi:10.1109/ICSE.2012.6227168.

———. 2013. “F3: fault localization for field failures.” In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, 213–23. New York, New York, USA: ACM Press. doi:10.1145/2483760.2483763.

Johnson, Brittany, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. “Why don’t software developers use static analysis tools to find bugs?” In *Proceedings of the International Conference on Software Engineering*, 672–81. IEEE Press. <http://dl.acm.org/citation.cfm?id=2486788.2486877>.

Johnson, J Howard. 1993. “Identifying redundancy in source code using fingerprints.” In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research*, 171–83. IBM Press. <http://dl.acm.org/citation.cfm?id=962289.962305>.

Johnson, J. Howard. 1994. “Visualizing textual redundancy in legacy source.” In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research*, 32. IBM Press. <http://dl.acm.org/citation.cfm?id=782185.782217>.

Joshua O’Madadhain, Danyel Fisher, Scott White, Padhraic Smyth, Yan-biao Boey. 2005. “Analysis and Visualization of Network Data using JUNG.” *Journal of Statistical Software* 10 (2): 1–35.

Juergens, Elmar, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner.

2009. “Do code clones matter?” In *Proceedings of the IEEE 31st International Conference on Software Engineering*, 485–95. IEEE. doi:10.1109/ICSE.2009.5070547.

Jung, Yungbum, Hakjoo Oh, and Kwangkeun Yi. 2009. “Identifying static analysis techniques for finding non-fix hunks in fix revisions.” In *Proceeding of the ACM First International Workshop on Data-Intensive Software Management and Mining - DSMM '09*, 13. New York, New York, USA: ACM Press. doi:10.1145/1651309.1651313.

Kamei, Yasutaka, Emad Shihab, Bram Adams, Ahmed E. Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. 2013. “A large-scale empirical study of just-in-time quality assurance.” *IEEE Transactions on Software Engineering* 39 (6). IEEE: 757–73. doi:10.1109/TSE.2012.70.

Kamiya, T., S. Kusumoto, and K. Inoue. 2002. “CCFinder: a multilinguistic token-based code clone detection system for large scale source code.” *IEEE Transactions on Software Engineering* 28 (7). IEEE Press: 654–70. doi:10.1109/TSE.2002.1019480.

Kapser, C., and M.W. Godfrey. 2004. “Aiding comprehension of cloning through categorization.” In *Proceedings. 7th International Workshop on Principles of Software Evolution, 2004.*, 85–94. IEEE. doi:10.1109/IWPSE.2004.1334772.

Kapser, Cory, and Michael Godfrey. 2006. “Cloning Considered Harmful Considered Harmful.” In *Proceedings of the 13th Working Conference on Reverse Engineering*, 19–28. IEEE. doi:10.1109/WCRE.2006.1.

Kapser, Cory, and Michael W Godfrey. 2003. “Toward a Taxonomy of Clones in Source Code: A Case Study.” In *International Workshop on Evolution of Large Scale Industrial Software Architectures*, 67–78. <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.5.6056>.

Kim, Dongsun, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. “Automatic patch generation learned from human-written patches.” In *Proceedings of the International Conference on Software Engineering*, 802–11. Ieee. doi:10.1109/ICSE.2013.6606626.

Kim, Dongsun, Xinming Wang, Student Member, Sunghun Kim, Andreas Zeller, S C Cheung, Senior Member, and Sooyong Park. 2011. “Which Crashes Should I Fix First?: Predicting Top Crashes at an Early Stage to Prioritize Debugging Efforts.” *TRANSACTIONS ON SOFTWARE ENGINEERING* 37 (3): 430–47.

Kim, Miryung, Vibha Sazawal, and David Notkin. 2005. “An empirical study of code clone genealogies.” *ACM SIGSOFT Software Engineering Notes* 30 (5). ACM:

187–96. doi:10.1145/1095430.1081737.

Kim, Sunghun, and Michael D Ernst. 2007a. “Which warnings should I fix first?” *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering ESECFSE 07*, 45. doi:10.1145/1287624.1287633.

Kim, Sunghun, and Michael D. Ernst. 2007b. “Prioritizing warning categories by analyzing software history.” In *Proceedings - ICSE 2007 Workshops: Fourth International Workshop on Mining Software Repositories, MSR 2007*. doi:10.1109/MSR.2007.26.

Kim, Sunghun, Kai Pan, and E. E. James Whitehead. 2006. “Memories of bug fixes.” In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering - SIGSOFT ’06/FSE-14*, 35. New York, New York, USA: ACM Press. doi:10.1145/1181775.1181781.

Kim, Sunghun, Thomas Zimmermann, Kai Pan, and E. Jr. Whitehead. 2006a. “Automatic Identification of Bug-Introducing Changes.” In *Proceedings of the International Conference on Automated Software Engineering*, 81–90. IEEE. doi:10.1109/ASE.2006.23.

———. 2006b. “Automatic Identification of Bug-Introducing Changes.” In *Proceedings of the International Conference on Automated Software Engineering*, 81–90. IEEE. doi:10.1109/ASE.2006.23.

Kim, Sunghun, Thomas Zimmermann, E. James Whitehead Jr., and Andreas Zeller. 2007a. “Predicting Faults from Cached History.” In *Proceedings of the International Conference on Software Engineering*, 489–98. IEEE. doi:10.1109/ICSE.2007.66.

———. 2007b. “Predicting Faults from Cached History.” In *Proceedings of the International Conference on Software Engineering*, 489–98. IEEE. doi:10.1109/ICSE.2007.66.

Ko, Andrew, Brad Myers, Michael Coblenz, and Htet Aung. 2006. “An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks.” *IEEE Transactions on Software Engineering* 32 (12): 971–87. doi:10.1109/TSE.2006.116.

Kontogiannis, K. 1997. “Evaluation experiments on the detection of programming patterns using software metrics.” In *Proceedings of the Fourth Working Conference on Reverse Engineering*, 44–54. IEEE Comput. Soc. doi:10.1109/WCRE.1997.624575.

Koponen, Timo. 2006. “Life cycle of defects in open source software projects.”

In *Open Source Systems*, 195–200. Springer.

Kpodjedo, Segla, Filippo Ricca, Philippe Galinier, Yann-Gal Guhneuc, and Giuliano Antoniol. 2010. “Design evolution metrics for defect prediction in object oriented systems.” *Empirical Software Engineering* 16 (1): 141–75. doi:10.1007/s10664-010-9151-7.

Krinke, Jens. 2001. “Identifying Similar Code with Program Dependence Graphs.” In *Proceedings of the Eighth IEEE Working Conference on Reverse Engineering*, 301–9. IEEE Computer Society. <http://dl.acm.org/citation.cfm?id=832308.837142>.

Kripke, Saul A. 1963. “Semantical Considerations on Modal Logic.” *Acta Philosophica Fennica* 16 (1963): 83–94.

Kropf, Thomas. 1999. *Introduction to formal hardware verification*. Springer.

Lamkanfi, Ahmed, Serge Demeyer, Emanuel Giger, and Bart Goethals. 2010. “Predicting the severity of a reported bug.” In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, 1–10. IEEE. doi:10.1109/MSR.2010.5463284.

Landis, J. Richard, and Gary G. Koch. 1977. “The Measurement of Observer Agreement for Categorical Data.” *Biometrics* 33 (1): 159. doi:10.2307/2529310.

Latoza, Thomas D., Gina Venolia, and Robert DeLine. 2006. “Maintaining mental models: a study of developer work habits.” In *Proceeding of the 28th International Conference on Software Engineering - ICSE '06*, 492–501. New York, New York, USA: ACM Press. doi:10.1145/1134285.1134355.

Layman, Lucas, Laurie Williams, and Robert St. Amant. 2007. “Toward Reducing Fault Fix Time: Understanding Developer Behavior for the Design of Automated Fault Detection Tools.” In *Proceedings of the First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*, 176–85. IEEE. doi:10.1109/ESEM.2007.11.

Le Goues, Claire, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. “A systematic study of automated program repair: Fixing 55 out of 105 bugs for \$8 each.” In *Proceedings of the International Conference on Software Engineering*, 3–13. IEEE.

Le, Xuan-Bach D, Tien-Duy B Le, and David Lo. 2015. “Should fixing these failures be delegated to automated program repair?” In *Proceedings of the International*

Symposium on Software Reliability Engineering, 427–37. IEEE.

Lee, Taek, Jaechang Nam, DongGyun Han, Sunghun Kim, and Hoh Peter In. 2011. “Micro interaction metrics for defect prediction.” In *Proceedings of the European Conference on Foundations of Software Engineering*, 311–231. New York, New York, USA: ACM Press. doi:10.1145/2025113.2025156.

Lewis, Chris, Zhongpeng Lin, Caitlin Sadowski, Xiaoyan Zhu, Rong Ou, and E. James Whitehead Jr. 2013. “Does bug prediction support human developers? Findings from a google case study.” In *Proceedings of the International Conference on Software Engineering*, 372–81. IEEE Press. <http://dl.acm.org/citation.cfm?id=2486788.2486838>.

Li, Z., S. Lu, S. Myagmar, and Y. Zhou. 2006. “CP-Miner: finding copy-paste and related bugs in large-scale software code.” *IEEE Transactions on Software Engineering* 32 (3). IEEE: 176–92. doi:10.1109/TSE.2006.28.

Lo, D. 2013. “A Comparative Study of Supervised Learning Algorithms for Reopened Bug Prediction.” In *Proceedings of the European Conference on Software Maintenance and Reengineering*, 331–34. IEEE. doi:10.1109/CSMR.2013.43.

Lopez, Nicolas, and Andr van der Hoek. 2011. “The code orb: supporting contextualized coding via at-a-glance views.” In *Proceeding of the 33rd International Conference on Software Engineering*, 824–27. New York, New York, USA: ACM Press. doi:10.1145/1985793.1985914.

Maiga, A., A. Hamou-Lhadj, M. Nayrolles, K. Koochekian Sabor, and A. Larson. 2015. “An empirical study on the handling of crash reports in a large software company: An experience report.” In *2015 IEEE 31st International Conference on Software Maintenance and Evolution, ICSME 2015 - Proceedings*, 342–51. doi:10.1109/ICSM.2015.7332485.

Manber, Udi. 1994. “Finding similar files in a large file system.” In *Proceedings of the Usenix Winter*, 1–10. USENIX Association. <http://dl.acm.org/citation.cfm?id=1267074.1267076>.

Manevich, Roman, Manu Sridharan, and Stephen Adams. 2004. “PSE: explaining program failures via postmortem static analysis.” In *ACM SIGSOFT Software Engineering Notes*, 29:63. 6. ACM. doi:10.1145/1041685.1029907.

Marcus, A., and J.I. Maletic. 2001. “Identification of high-level concept clones in source code.” In *Proceedings International Conference on Automated Software*

Engineering, 107–14. IEEE Comput. Soc. doi:10.1109/ASE.2001.989796.

Martin Fowler. 2009. “FeatureBranch.” <http://martinfowler.com/bliki/FeatureBranch.html>.

McMillan, Kenneth L. 1993. *Symbolic model checking*. Springer.

Menzies, Tim, and Andrian Marcus. 2008. “Automated severity assessment of software defect reports.” In *2008 IEEE International Conference on Software Maintenance*, 346–55. IEEE. doi:10.1109/ICSM.2008.4658083.

Menzies, Tim, Alex Dekhtyar, Justin Distefano, and Jeremy Greenwald. 2007. “Problems with Precision: A Response to ‘Comments on’ Data Mining Static Code Attributes to Learn Defect Predictors.” *IEEE Transactions on Software Engineering* 33 (9). IEEE Computer Society: 637.

Moha, N., Y.-G. Gueheneuc, L. Duchien, and A.-F. Le Meur. 2010. “DECOR: A Method for the Specification and Detection of Code and Design Smells.” *IEEE Transactions on Software Engineering* 36 (1): 20–36. doi:10.1109/TSE.2009.50.

Moha, Naouel; Francis; Palma, Mathieu; Nayrolles, Benjamin; Joyen-Conseil, Yann-Gal; Guhneuc, Benoit; Baudry, and Jean-Marc; Jzquel. 2012. “Specification and Detection of SOA Antipatterns.” *Proceedings of the International Conference on Service Oriented Computing*. Springer, 1–16.

Nagappan, N., and T. Ball. 2005. “Use of relative code churn measures to predict system defect density.” In *Proceedings of the International Conference on Software Engineering*, 284–92. IEEE. doi:10.1109/ICSE.2005.1553571.

Nagappan, Nachiappan, and Thomas Ball. 2005. “Static analysis tools as early indicators of pre-release defect density.” In *Proceedings of the International Conference on Software Engineering*, 580–86. New York, New York, USA: ACM Press. doi:10.1145/1062455.1062558.

Nagappan, Nachiappan, Thomas Ball, and Andreas Zeller. 2006. “Mining metrics to predict component failures.” In *Proceeding of the International Conference on Software Engineering*, 452–61. New York, New York, USA: ACM Press. doi:10.1145/1134285.1134349.

Nam, Jaechang, Sinno Jialin Pan, and Sunghun Kim. 2013. “Transfer defect learning.” In *Proceedings of the International Conference on Software Engineering*, 382–91. Ieee. doi:10.1109/ICSE.2013.6606584.

Narayanasamy, Satish, Gilles Pokam, and Brad Calder. 2005. “BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging.” In

Proceedings of the 32nd Annual International Symposium on Computer Architecture, 33:284–95. 2. ACM. doi:10.1145/1080695.1069994.

NASA. 2009. “Open Mission Control Technologies.” <https://sites.google.com/site/openmct/>.

Nayrolles, M., and W. Hamou-Lhadj. 2016. “BUMPER: A Tool to Cope with Natural Language Search of Millions Bugs and Fixes.” In *International Conference on Software Analysis, Evolution, and Reengineering*, 649–52. IEEE.

Nayrolles, M., A. Hamou-Lhadj, S. Tahar, and A. Larsson. 2015. “JCHARMING: A bug reproduction approach using crash traces and directed model checking.” In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015 - Proceedings*. doi:10.1109/SANER.2015.7081820.

Nayrolles, Mathieu, and Adblewahab Hamou-lhadj. 2018. “CLEVER: Combining Code Metrics with Clone Detection for Just-In-Time Fault Prevention and Resolution in Large Industrial Projects.” In *Mining Software Repositories*.

Nayrolles, Mathieu, Eric Beaudry, Naouel Moha, and Petko Valtchev. 2015. “Towards Quality-Driven SOA Systems Refactoring through Planning.” In *6th International MCETECH Conference*.

Nayrolles, Mathieu, Abdelwahab Hamou-Lhadj, Tahar Sofiene, and Alf Larsson. 2015. “JCHARMING : A Bug Reproduction Approach Using Crash Traces and Directed Model Checking.” In *Proceedings of the 22nd International Conference on Software Analysis, Evolution, and Reengineering*, 101–10.

Nayrolles, Mathieu, Abdelwahab Hamou-Lhadj, Sofine Tahar, and Alf Larsson. 2016. “A bug reproduction approach based on directed model checking and crash traces.” *Journal of Software: Evolution and Process*. doi:10.1002/smr.1789.

Nayrolles, Mathieu, Abdou Maiga, Abdelwahab Hamou-lhadj, and Alf Larsson. n.d. “A Taxonomy of Bugs : An Empirical Study,” 1–10.

Nayrolles, Mathieu, Naouel Moha, and Petko Valtchev. 2013. “Improving SOA Antipatterns Detection in Service Based Systems by Mining Execution Traces.” In *Working Conference on Reverse Engineering*, 321–30. i. IEEE.

Nayrolles, Mathieu; 2013. “Improving SOA Antipattern Detection in Service Based Systems by Mining Execution Traces.” PhD thesis.

Nessa, Syeda, Muhammad Abedin, W Eric Wong, Latifur Khan, and Yu Qi. 2008.

“Software Fault Localization Using N -gram Analysis.” In *WASA*, 548–59.

Newman, M. E. J., and M. Girvan. 2004. “Finding and evaluating community structure in networks.” *Physical Review E* 69 (2). American Physical Society: 026113. doi:10.1103/PhysRevE.69.026113.

Nguyen, Anh Tuan, Tung Thanh Nguyen, Tien N. Nguyen, David Lo, and Chengnian Sun. 2012. “Duplicate bug report detection with a combination of information retrieval and topic modeling.” In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering - ASE 2012*, 70. New York, New York, USA: ACM Press. doi:10.1145/2351676.2351687.

Nguyen, Hoan Anh, Tung Thanh Nguyen, Nam H. Pham, Jafar Al-Kofahi, and Tien N. Nguyen. 2012. “Clone Management for Evolving Software.” *IEEE Transactions on Software Engineering* 38 (5): 1008–26. doi:10.1109/TSE.2011.90.

Nguyen, Tung Thanh, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. 2009. “Clone-Aware Configuration Management.” In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, 123–34. IEEE. doi:10.1109/ASE.2009.90.

Niko, Schwarz, Lungu Mircea, and Robbes Romain. 2012. “On how often code is cloned across repositories.” In *Proceedings of the 34th International Conference on Software Engineering*, 1289–92. IEEE.

Norman, Donald A. 2013. *The Design of Everyday Things: Revised and Expanded Edition*. Basic Books. <https://books.google.com/books?hl=en&lr=&id=nVQPAAAAQBAJ&pg>

Northrop, Linda, Peter Feiler, Richard P Gabriel, John Goodenough, Rick Linger, Tom Longstaff, Rick Kazman, et al. 2006. “Ultra-large-scale systems: The software challenge of the future.” DTIC Document.

O’Sullivan, Bryan, and Bryan. 2009. “Making sense of revision-control systems.” *Communications of the ACM* 52 (9). ACM: 56. doi:10.1145/1562164.1562183.

Object Refinery Limited. 2005. “JFreeChart.” <http://jfree.org/jfreechart/>.

Oracle. 2011. “Throwable (Java Platform SE6).” <http://docs.oracle.com/javase/6/docs/api/java/lang/Throwable.html>.

Ostrand, T.J., E.J. Weyuker, and R.M. Bell. 2005. “Predicting the location and number of faults in large software systems.” *IEEE Transactions on Software*

Engineering 31 (4). IEEE: 340–55. doi:10.1109/TSE.2005.49.

Palma, Francis. 2013. “Detection of SOA Antipatterns.” PhD thesis, Ecole Polytechnique de Montreal.

Palma, Francis, Mathieu Nayrolles, and Naouel Moha. 2013. “SOA Antipatterns : An Approach for their Specification and Detection.” *International Journal of Cooperative Information Systems* 22 (04): 1–40.

Pan, Kai, Sunghun Kim, and E. James Whitehead. 2008. “Toward an understanding of bug fix patterns.” *Empirical Software Engineering* 14 (3): 286–315. doi:10.1007/s10664-008-9077-5.

Patenaude, J.-F., E. Merlo, M. Dagenais, and B. Lague. 1999. “Extending software quality assessment techniques to Java systems.” In *Proceedings Seventh International Workshop on Program Comprehension*, 49–56. IEEE Comput. Soc. doi:10.1109/WPC.1999.777743.

Perry, Dewayne E., and Carol S. Stieg. 1993. “Software faults in evolving a large, real-time system: a case study.” In *Software Engineering—ESEC*, 48–67.

Pratt, Michael J. 2001. “Introduction to ISO 10303—the STEP Standard for Product Data Exchange.” *Journal of Computing and Information Science in Engineering* 1 (1). American Society of Mechanical Engineers: 102. doi:10.1115/1.1354995.

Pressman, Roger S. 2005. *Software Engineering: A Practitioner’s Approach*. Palgrave Macmillan. <https://books.google.com/books?hl=en&lr=&id=bL7QZHtWvaUC&pgis=1>.

Radatz, Jane, Anne Geraci, and Freny Katki. 1990. “IEEE standard glossary of software engineering terminology.” *IEEE Std* 610121990 (121990): 3.

Rahman, Foyzur, and Premkumar Devanbu. 2013. “How, and why, process metrics are better.” In *Proceedings of the International Conference on Software Engineering*, 432–41. IEEE Press.

Robertson, T J, Shrinu Prabhakararao, Margaret Burnett, Curtis Cook, Joseph R Ruthruff, Laura Beckwith, and Amit Phalgune. 2004. “Impact of interruption style on end-user debugging.” In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 287–94. doi:10.1145/985692.985729.

Robertson, T. J., Joseph Lawrance, and Margaret Burnett. 2006. “Impact of high-intensity negotiated-style interruptions on end-user debugging.” *Journal of Visual*

Languages and Computing 17 (2): 187–202. doi:10.1016/j.jvlc.2005.09.002.

Roehm, Tobias, Stefan Nosovic, and Bernd Bruegge. 2015. “Automated extraction of failure reproduction steps from user interaction traces.” In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 121–30. IEEE. doi:10.1109/SANER.2015.7081822.

Rosen, Christoffer, Ben Grawi, and Emad Shihab. 2015. “Commit guru: analytics and risk prediction of software commits.” In *Proceedings of the Joint Meeting on Foundations of Software Engineering*, 966–69. New York, New York, USA: ACM Press. doi:10.1145/2786805.2803183.

Rler, Jeremias, Andreas Zeller, Gordon Fraser, Cristian Zamfir, and George Candea. 2013. “Reconstructing Core Dumps.” In *Proceedings of the 6th International Conference on Software Testing, Verification and Validation, Ser. ICST*.

Roy, Chanchal K. 2009. “Detection and Analysis of Near-Miss Software Clones.” PhD thesis, Queen’s University.

Roy, Chanchal K., and James R. Cordy. 2008. “An Empirical Study of Function Clones in Open Source Software.” In *Proceedings of the Working Conference on Reverse Engineering*, 81–90. IEEE. doi:10.1109/WCRE.2008.54.

Runeson, Per, Magnus Alexandersson, and Oskar Nyholm. 2007. “Detection of Duplicate Defect Reports Using Natural Language Processing.” In *29th International Conference on Software Engineering*, 499–510. IEEE. doi:10.1109/ICSE.2007.32.

Saha, Ripon K., Sarfraz Khurshid, and Dewayne E. Perry. 2014. “An empirical study of long lived bugs.” In *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, 144–53. IEEE. doi:10.1109/CSMR-WCRE.2014.6747164.

Saini, Vaibhav, Hitesh Sajnani, Jaewoo Kim, and Cristina Lopes. 2016. “SourcererCC and SourcererCC-I: Tools to Detect Clones in Batch mode and During Software Development.” In *Proceedings of the 38th International Conference on Software Engineering*, 597–600. New York, New York, USA: ACM Press. doi:10.1145/2889160.2889165.

Sajnani, Hitesh, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. 2016. “SourcererCC: scaling code clone detection to big-code.” In *Proceedings of the 38th International Conference on Software Engineering*, 1157–68. New

York, New York, USA: ACM Press. doi:10.1145/2884781.2884877.

Seinturier, Lionel, Philippe Merle, Romain Rouvoy, Daniel Romero, Valerio Schiavoni, and Jean-Bernard Stefani. 2012. “A component-based middleware platform for reconfigurable service-oriented architectures.” *Software: Practice and Experience* 42 (5): 559–83. doi:10.1002/spe.1077.

Shen, Haihao, Jianhong Fang, and Jianjun Zhao. 2011. “EFindBugs: Effective Error Ranking for FindBugs.” In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, 299–308. IEEE. doi:10.1109/ICST.2011.51.

Shihab, Emad, Akinori Ihara, Yasutaka Kamei, Walid M. Ibrahim, Masao Ohira, Bram Adams, Ahmed E. Hassan, and Ken-ichi Matsumoto. 2010. “Predicting Reopened Bugs: A Case Study on the Eclipse Project.” In *2010 17th Working Conference on Reverse Engineering*, 249–58. IEEE. doi:10.1109/WCRE.2010.36.

Shivaji, Shivkumar, Student Member, Senior Member, Ram Akella, and Sunghun Kim. 2013. “Reducing Features to Improve Code Change-Based Bug Prediction.” *IEEE Transactions on Software Engineering* 39 (4): 552–69.

Snyder, Bruce, Dejan Bosanac, and Rob Davies. 2011. *ActiveMQ in Action*. Manning Publications Co.

Steven, John, Pravir Chandra, Bob Fleck, and Andy Podgurski. 2000. “jRapture: A Capture/Replay Tool for Observation-Based Testing.” In *Proceedings of the International Symposium on Software Testing and Analysis.*, 158–67. August.

Subramanyam, R., and M.S. Krishnan. 2003. “Empirical analysis of CK metrics for object-oriented design complexity: implications for software defects.” *IEEE Transactions on Software Engineering* 29 (4). IEEE: 297–310. doi:10.1109/TSE.2003.1191795.

Sun, Chengnian, David Lo, Siau-Cheng Khoo, and Jing Jiang. 2011. “Towards more accurate retrieval of duplicate bug reports.” *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, November. Ieee, 253–62. doi:10.1109/ASE.2011.6100061.

Sunghun Kim, Sunghun, E.J. Whitehead, and Yi Yi Zhang. 2008. “Classifying Software Changes: Clean or Buggy?” *IEEE Transactions on Software Engineering* 34 (2). IEEE: 181–96. doi:10.1109/TSE.2007.70773.

Suzaki, Kiyoshi. 1987. *New manufacturing challenge: Techniques for continuous improvement*. Simon; Schuster.

Tairas, Robert, Jeff Gray, and Ira Baxter. 2006. “Visualization of clone detection

results.” In *Proceedings of the 2006 OOPSLA Workshop on Eclipse Technology Exchange*, 50–54. New York, New York, USA: ACM Press. doi:10.1145/1188835.1188846.

Tamrawi, Ahmed, Tung Thanh Nguyen, Jafar Al-Kofahi, and Tien N. Nguyen. 2011. “Fuzzy set-based automatic bug triaging.” In *Proceeding of the 33rd International Conference on Software Engineering - ICSE '11*, 884. New York, New York, USA: ACM Press. doi:10.1145/1985793.1985934.

Tao, Yida, Jindae Kim, Sunghun Kim, and Chang Xu. 2014. “Automatically generated patches as debugging aids: a human study.” In *Proceedings of the International Symposium on Foundations of Software Engineering*, 64–74. ACM.

The Apache Software Foundation. 1999. “Log4j 2 Guide - Apache Log4j 2.” <http://logging.apache.org/log4j/2.x/>.

———. 2015. “Apache BatchEE.” In [Http://batchee.incubator.apache.org/](http://batchee.incubator.apache.org/). <http://batchee.incubator.apache.org/>.

Tian, Yuan, David Lo, and Chengnian Sun. 2012. “Information Retrieval Based Nearest Neighbor Classification for Fine-Grained Bug Severity Prediction.” In *2012 19th Working Conference on Reverse Engineering*, 215–24. IEEE. doi:10.1109/WCRE.2012.31.

Tian, Yuan, Chengnian Sun, and David Lo. 2012. “Improved Duplicate Bug Report Identification.” In *2012 16th European Conference on Software Maintenance and Reengineering*, 385–90. IEEE. doi:10.1109/CSMR.2012.48.

Tin Kam Ho. 1995. “Random decision forests.” In *Proceedings of the International Conference on Document Analysis and Recognition*, 1:278–82. IEEE Comput. Soc. Press. doi:10.1109/ICDAR.1995.598994.

———. 1998. “The random subspace method for constructing decision forests.” *IEEE Transactions on Pattern Analysis and Machine Intelligence* 20 (8): 832–44. doi:10.1109/34.709601.

Toomey, Warren. 2012. “Ctcompare: Code clone detection using hashed token sequences.” In *Proceedings of the 6th International Workshop on Software Clones*, 92–93. IEEE. doi:10.1109/IWSC.2012.6227881.

Tourani, Parastou, and Bram Adams. 2016. “The Impact of Human Discussions on Just-in-Time Quality Assurance: An Empirical Study on OpenStack and Eclipse.” In *23rd International Conference on Software Analysis, Evolution, and Reengineering*, 189–200. IEEE. doi:10.1109/SANER.2016.113.

Tufano, Michele, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano

Di Penta, Andrea De Lucia, and Denys Poshyvanyk. 2015. “When and why your code starts to smell bad,” May. IEEE Press, 403–14. <http://dl.acm.org/citation.cfm?id=2818754.2818805>.

Valdivia Garcia, Harold, and Emad Shihab. 2014. “Characterizing and predicting blocking bugs in open source projects.” In *Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014*, 72–81. New York, New York, USA: ACM Press. doi:10.1145/2597073.2597099.

Visser, Willem, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. 2003. “Model Checking Programs.” In *Automated Software Engineering*, 10:203–32. 2. Springer.

Visser, Willem, Corina S. Psreanu, and Sarfraz Khurshid. 2004. “Test input generation with java PathFinder.” *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*. New York, New York, USA: ACM Press, 97. doi:10.1145/1007512.1007526.

Wahler, V., D. Seipel, J. Wolff, and G. Fischer. 2004. “Clone detection in source code by frequent itemset techniques.” In *Proceedings of the Fourth IEEE International Workshop on Source Code Analysis and Manipulation*, 128–35. IEEE Comput. Soc. doi:10.1109/SCAM.2004.6.

Wang, Xinlei (Oscar), Eilwoo Baik, and Premkumar T. Devanbu. 2011. “System compatibility analysis of Eclipse and Netbeans based on bug data.” In *Proceeding of the 8th Working Conference on Mining Software Repositories - MSR '11*, 230. New York, New York, USA: ACM Press. doi:10.1145/1985441.1985479.

Weiss, Cathrin, Rahul Premraj, Thomas Zimmermann, and Andreas Zeller. 2007. “How Long Will It Take to Fix This Bug?” In *Fourth International Workshop on Mining Software Repositories (MSR'07:ICSE Workshops 2007)*, 1–1. IEEE. doi:10.1109/MSR.2007.13.

Wei, Cathrin, Thomas Zimmermann, and Andreas Zeller. 2007. “How Long will it Take to Fix This Bug ?” In *Fourth International Workshop on Mining Software Repositories (MSR'07)*, 1. 2.

Wellington, Brian. 2013. “Dnsjava.” <http://www.dnsjava.org/>.

Wettel, R., and R. Marinescu. 2005. “Archeology of code duplication: recovering duplication chains from small duplication fragments.” In *Proceedings of the Seventh International Symposium on Symbolic and Numeric Algorithms for Scientific*

Computing, 63–71. IEEE. doi:10.1109/SYNASC.2005.20.

Whittaker, James A., Jason Arbon, and Jeff Carollo. 2012. *How Google Tests Software*. Addison-Wesley. <https://books.google.com/books?hl=en&lr=&id=VrAx1ATf-RoC&pgis=1>.

Wu, Rongxin, Hongyu Zhang, Sunghun Kim, and SC Cheung. 2011. “Relink: recovering links between bugs and changes.” In *Proceedings of the European Conference on Foundations of Software Engineering*, 15–25. <http://dl.acm.org/citation.cfm?id=2025120>.

Xuan, Jifeng, He Jiang, Zhilei Ren, and Weiqin Zou. 2012. “Developer prioritization in bug repositories.” In *2012 34th International Conference on Software Engineering (ICSE)*, 25–35. IEEE. doi:10.1109/ICSE.2012.6227209.

Yamanaka, Yuki, Eunjong Choi, Norihiro Yoshida, Katsuro Inoue, and Tateki Sano. 2012. “Industrial application of clone change management system.” In *Proceedings of the 6th International Workshop on Software Clones*, 67–71. IEEE Press.

Yang, Xinli, David Lo, Xin Xia, Yun Zhang, and Jianling Sun. 2015. “Deep Learning for Just-in-Time Defect Prediction.” In *2015 IEEE International Conference on Software Quality, Reliability and Security*, 17–26. IEEE. doi:10.1109/QRS.2015.14.

Zamfir, Cristian, and George Candea. 2010. “Execution Synthesis: A Technique for Automated Software Debugging.” In *Proceedings of the 5th European Conference on Computer Systems*, 321–34.

Zeller, A., and R. Hildebrandt. 2002. “Simplifying and isolating failure-inducing input.” *IEEE Transactions on Software Engineering* 28 (2). IEEE Press: 183–200. doi:10.1109/32.988498.

Zeller, Andreas. 1997. *Configuration management with version sets: A unified software versioning model and its applications*.

———. 2013. “Where Should We Fix This Bug? A Two-Phase Recommendation Model.” *IEEE Transactions on Software Engineering* 39 (11). IEEE: 1597–1610. doi:10.1109/TSE.2013.24.

Zhang, Feng, Foutse Khomh, Ying Zou, and Ahmed E. Hassan. 2012. “An Empirical Study on Factors Impacting Bug Fixing Time.” In *2012 19th Working Conference on Reverse Engineering*, 225–34. IEEE. doi:10.1109/WCRE.2012.32.

Zhang, Gang, Xin Peng, Zhenchang Xing, Shihai Jiang, Hai Wang, and Wenyun

Zhao. 2013. “Towards contextual and on-demand code clone management by continuous monitoring.” In *28th IEEE/ACM International Conference on Automated Software Engineering*, 497–507. IEEE. doi:10.1109/ASE.2013.6693107.

Zhang, Hongyu, Liang Gong, and Steve Versteeg. 2013. “Predicting bug-fixing time: an empirical study of commercial software projects.” In *International Conference on Software Engineering*, 1042–51. IEEE Press. <http://dl.acm.org/citation.cfm?id=2486788.2486931>.

Zhou, Jian, Hongyu Zhang, and David Lo. 2012. “Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports.” In *Proceedings of the 34th International Conference on Software Engineering, IEEE*, 14–24. IEEE. doi:10.1109/ICSE.2012.6227210.

Zibran, Minhaz F., and Chanchal K. Roy. 2011. “Towards flexible code clone detection, management, and refactoring in IDE.” In *Proceeding of the 5th International Workshop on Software Clones*, 75–76. New York, New York, USA: ACM Press. doi:10.1145/1985404.1985423.

———. 2012. “IDE-based real-time focused search for near-miss clones.” In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, 1235–42. New York, New York, USA: ACM Press. doi:10.1145/2245276.2231970.

Zimmermann, Thomas, and Nachiappan Nagappan. 2008. “Predicting defects using network analysis on dependency graphs.” In *Proceedings of the International Conference on Software Engineering*, 531. New York, New York, USA: ACM Press. doi:10.1145/1368088.1368161.

Zimmermann, Thomas, Nachiappan Nagappan, Philip J. Guo, and Brendan Murphy. 2012. “Characterizing and predicting which bugs get reopened.” In *Proceedings of the 34th International Conference on Software Engineering, IEEE*, 1074–83. IEEE. doi:10.1109/ICSE.2012.6227112.

Zimmermann, Thomas, Rahul Premraj, and Andreas Zeller. 2007. “Predicting Defects for Eclipse.” In *Proceedings of the International Workshop on Predictor Models in Software Engineering*, 9. IEEE. doi:10.1109/PROMISE.2007.10.

Zuddas, Daniele, Wei Jin, Fabrizio Pastore, Leonardo Mariani, and Alessandro Orso. 2014. “MIMIC: locating and understanding bugs by analyzing mimicked executions.” In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, 815–26. New York, New York, USA: ACM Press.

doi:10.1145/2642937.2643014.