TOOLS AND TECHNIQUES TO IMPROVE SOFTWARE MAINTENANCE AT COMMIT-TIME

Mathieu Nayrolles

A research proposal
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

March 2016

Abstract

Tools and techniques to improve software maintenance at commit-time

Mathieu Nayrolles, Ph.D.
Concordia University, 2016

The maintenance and evolution of complex software systems account for more than 70% software's life cycle. More than two decades of research have been conducted to improve our knowledge of these processes in terms of issue triaging, issue prediction, duplicate issue detection, issue reproduction and changes prediction. This research gave meaning to the millions of issues that can be found in project and revision management systems. Context-aware IDE and think tank in open source architecture, openned the path to approaches that support developers during their programming sessions by leveraging past knowledge and architectures.

However, these techniques are still not broadly adopted by praticionners in large software companies. The main problem is the lack of actionable inteligence. Indeed, such systems tend to be seen as black boxes that yield false positive by end-users. Moreover, actions to resolve an identified problem can be hard to identify or to apply.

In this research proposal, we present four approaches: (a) an online bug-fix search engine and API (`BUMPER`, Bug MetarePository for dEveloper and Researcher), (b) an approach to automatically reproduce bug submitted to bug-report systems (`JCHARMING`, Java CrasH Automatic Reproduction by directed Model checkING), (c) a recommendation system to propose the right auto-completion at the right time using actionnable inteligence (`RESEMBLE`, REcommendation System based on cochangE Mining at Block LEvel) and, finally, (d) an approach to prevent bug insertion at commit-time by leveraging decade of open-source history (`BIANCA`, Bug Insertion ANticipation by Clone Analysis at commit time). We also propose a taxonomy of bugs. When combined into `pErICOPE` (Ecosystem Improve source COde during Programming session with real-time mining of common knowlEdge), these tools (i) provide the possibility to search related software artifacts using natural language, (ii) accurately reproduce field-crash in lab environment, (iii) recommend improvement or completion of block of code under edition and (iv) prevent the introduction of issues at commit time.

This proposal develops these ideas while highlighting remaining issues and the PhD schedule.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Maintenance activities are known to be costly and challenging [Pre05]. Studies have shown that the cost of software maintenance can reach up to 70% of the overall cost of the software development process [HR02].
Lehman's laws of software evolution apply to three different types of software S, P and E [Leh80]. S-softwares are written according to an exact specification of what a program can do, E-softwares are written to perform some real-world activity; how it should behave is strongly linked to the environment in which it runs, and such a program needs to adapt to varying requirements and circumstances in that environment and, finally, P-softwares are programs written to implement certain procedures that completely determine what the program can do.

- "Continuing Change" — an E-type system must be continually adapted or it becomes progressively less satisfactory.
- "Increasing Complexity" — as an E-type system evolves, its complexity increases unless work is done to maintain or reduce it.
- "Self Regulation" — E-type system evolution processes are self-regulating with the distribution of product and process measures close to normal.
- "Conservation of Organisational Stability (invariant work rate)" - the average effective global activity rate in an evolving E-type system is invariant over the product's lifetime.
- "Conservation of Familiarity" — as an E-type system evolves, all associated with it, developers, sales personnel and users, for example, must maintain mastery of its content and behavior to achieve satisfactory evolution. Excessive growth diminishes

that mastery. Hence the average incremental growth remains invariant as the system evolves.

- "Continuing Growth" — the functional content of an E-type system must be continually increased to maintain user satisfaction over its lifetime.
- "Declining Quality" — the quality of an E-type system will appear to be declining unless it is rigorously maintained and adapted to operational environment changes.
- "Feedback System" — E-type evolution processes constitute multi-level, multi-loop, multi-agent feedback systems and must be treated as such to achieve significant improvement over any reasonable base.

While these laws, written between 1974 and 1996, still very much apply to nowadays software engineering, engineers and practitioners have created tools, techniques and processes to control their negative impacts. For instance, most of real world project's—in opposition to test / very small size projects—host their source code on a version control system [Roc75] that is able to keep track of the different revisions of a system and of the different changes made by the developers of the team. Version control systems help to control the *continuing change*, *increasing complexity* and *continuing growth* rules' effects. In order to manage the *feedback systems* and *declining quality* rules, organizations use issue & project tracking system to assign tasks to developers, report unexpected behaviors or crashes and track advancement.

In the last decade, source code revision control system and issue & tracking systems have grown to contain hundreds of thousands of revision, issues and tasks per project. Naturally, this plethora of data pushed researchers across the world to conduct hundreds of studies in several active research fields: bug reproduction, bug triaging, duplicated bug identification, bug comprehension, bug re-production.

Mining issue and source code version control systems is perhaps one of the most active research field today. The reason is that their analysis provides useful insight that can help with many maintenance activities such as bug fixing [WPZZ07, SKP14], bug reproduction [AKE08, JO12, Che13], fault analysis [JLL+12, JO13], etc. This increase of attention can be further justified by the emergence of many open source bug tracking systems, allowing open source software teams to make their bug reports available online to researchers.

1.1 Preliminaries

In this section, we explain what version control systems (in 1.1.1) and Issue & project tracking system (in 1.1.2) are. If one were to be familiar with Svn[1], Git[2], Mercurial[3], Bugzilla[4], Jira[5] and Github[6], one might want to skip this section.

1.1.1 Version control systems

Version control consists in maintaining the versions of files — such as source code and other software artifacts. This activity is extremely complex and cannot be done by hand on real world project[7]. Consequently, numerous softwares have been created to help practitioners manage the version of their software artifacts. Each evolution of a software is a version[8] (or revision) and each version (revision) is linked to the one before through modifications of software artifacts. These modifications consist in updating, adding or deleting software artifacts. They can be referred as `diff`, `patch` or `commit` [9]. Each `diff`, `patch` or `commit` have the following characteristics:

- Number of Files: The number of software artifacts that have been modified, added or deleted.
- Number of Hunks: The number of consecutive blocks of modified, added or deleted lines in textual files. Hunks are useful to determine, in each file, how many different places the developer has modified.
- Number of Churns: The number of lines modified. However, the churn value for a line change should be at least two as the line had to be deleted first and then added back with the modifications.

Providers

In this document, we will 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

[1] https://subversion.apache.org/
[2] https://git-scm.com/
[3] https://mercurial.selenic.com/
[4] https://www.bugzilla.org/
[5] https://www.atlassian.com/software/jira
[6] https://github.com/
[7] Once again, real world project qualifies projects that are done in an industrial environment rather than school or so.
[8] Software version is not to be confused with the version of a software which refer to the shipping of a final product to customers.
[9] These names are not to be used interchangeably as difference exists.

concurrent version system that can handle conflict in the different versions of different developers and it is widely used. 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 share a lot of concepts with `Svn`. Consequently, it will be easier for people used to `Svn` to switch to a distributed revision system if they use `Mercurial`.

1.1.2 Issue & Project Tracking Systems

Issue & project tracking systems allow end-users to directly create bug reports (BRs) to report on system crashes and manager can create tasks to drive the evolution forward. These systems are also used by development teams to manage the BRs, and keep track of the fixes.
The life cycle of an issue is as follows: After an issue is submitted by an end-user, 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 bug is then assigned to a developer to be fixed. When the bug is in the `ASSIGNED` state, the assigned developer(s) starts working on the issue. A fixed issue moves to the `RESOLVED` state. Developers have five different possibilities to resolve an issue: `FIXED`, `DUPLICATE`, `WONTFIX`, `WORKSFORME` and `INVALID`.

- `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 issue.
- `RESOLVED/DUPLICATE`: A previously submitted issue is being processed. The bug is marked as duplicate of the original bug.
- `RESOLVED/WONTFIX`: This is applied in the case where developers decide that a given bug will not be fixed.
- `RESOLVED/WORKSFORME`: If the issue cannot be reproduced on the reported OS / hardware.
- `RESOLVED/INVALID`: If the issue is not related to the software itself.

Finally, the issue is `CLOSED` after it is resolved. An issue 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 issue marked as the new one and the resolved status are known as the *fixing time*, usually in days. If the issue is reopened: the days

between the time the issue is reopened and the time it is marked again as `RESOLVED/FIXED` are cumulated. Issues can be reopened many times.
Tasks, follow a similar life cycle at the exception of the `UNCONFIRMED` and `RESOLVED` states. Tasks are created by management and do not need to be confirmed in order 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. Issues are considered as problems to eradicate in the program and thus, fold into the maintenance activity. Tasks are considered as new features or amelioration to include in the program and fold into the evolution activity.
Issues and tasks can (and must according to [BJS+08]) have a severity. The severity is a classification of a issue to indicate the degree of negative impact on the quality of software and can evolve at any point during the lifecycle of the bug. 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 issue, some loss of functionality under specific circumstances), minor (minor loss of function, or other problem where easy workaround is present) trivial (cosmetic problem like misspelled words or misaligned text).
The relationship between an issue or task and the actual modification can be hard to establish and it has been a subject of various research studies (e.g., [ACC+02, BBR+10, WZKC11]) for the simple reason that they are in two different systems: the revision and the issue management systems. While it is considered a good practice to link each issue with the source code revision system by indicating the issue $\#id$ on the modification message, more than half of the issues are not linked to a modification.

Providers

In this study, we collect data from four different bug tracking systems: *Bugzilla*, *Jira*, *Github* and *Sourceforge*. *Bugzilla* belongs to the Mozilla foundation and has first been released in 1998. *Jira*, provided by Altassian, has been released 14 years ago, in 2002. *Bugzilla* is 100% open source and it's difficult to estimate how many project uses it. However, we can, without any risks envision that it owns a great share of the market as major organizations such as Mozilla, Eclipse and the Apache Software Foundation uses it. *Jira*, in the other hand, is a commercial software — with a freemium business model — and Altassian 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 revision system and evolve, later on, to add issue and project management capabilities to their softwares. Nevertheless, this common particularity has the advantage to ease the link between issues and source code.

1.2 Motivation

Architects, the ones that design buildings — where mistakes cost lives — spend at least five years at school and possibly their whole carriers to study, understand and reproduce great designs made by great architects. Software architects, however, begin in programing 101 by displaying the famous "`Hello World`" statement and exponentially increase the complexity of their programs over their years of study and work. At some point, they will earn the title of software architect (or technical leader) because they have designed, maintained and evolved *enough* programs to be trustworthy on the matter. However, unlike building architects, they have to learn how to recognize, analyze and reproduce great architectural choices by themselves in addition of their day to day work. Of course, software developers do learn good practices such as design patterns [GHJV08] but in a very few occasions they will be presented with a state-of-the-art program built by great developers (Amy Brown *et al.* propose exactly that in their books [AW12, BG12, Arm13]).

While this research is not about reforming how programming classes are taught, we still want to ease the access to this knowledge for developers during their programming sessions in order to ship better programs.

In this research proposal, we shift the focus from merely mining revision and issue management system, where knowledge of great developers lies, to integrate them in their rightful place: as the keystone of software development and evolution activities. Extracting the ground truth from repositories helped engineers and practitioners to be better at building softwares as they know, for example, *how long it will take to fix a bug* [WPZZ07], *what makes a good bug report* [BJS+08] or *how to fix long-lived bugs* [SKP14]. Using these discoveries, tools can be created, on a per organization basis, to fit particular requirements such as programming languages, development processes or particular threshold. If we want to truthfully and deeply modify the software engineering landscape to have better softwares in terms of quality, maintainability and availability, we need to provide this information during the development, maintenance and evolution processes according to a specific context in an easy, reliable, actionable, free way.

If we look back at the history of software engineering, the increase of processors' speed and decrease of their price allowed one to have a compiler on its own machine rather than sending one's code to the mainframe and receive compilation errors hours (days) later. This allowed, among other factors, the democratization of software engineering as *everyone*, belonging to a major organization or not, became able to build code. We believe that, it is now time to allow developers, engineering and practitioners, regardless of their programming language and contextual environment, not only to write and build code but to write and

built qualitative, robust, resilient, easy to maintain and to fix code. What better way to do so than to *stand on the shoulder of giants* by having access to all the open sources repositories, including but not limited to, issues, tasks, bug fixes, patches, comments, good practices break down to the right level and provided at the right time during day to day programming sessions?
What we concretely propose is an open-source, free, automatable tool suite that will allow everyone to (1) automatically reproduce field crashes in a controlled lab environment without any privacy concerns, (2) search in natural language the issues, comments, bugs, patches and fixes of tens of thousands of open source project, (3) prevent the insertion of defects in the source code during programming sessions by providing examples of the similar defect and how it has been fixed with a programming language abstraction. To support these tools we propose an empirical bug taxonomy (4).

1.3 Thesis Contributions

In this section we present the problems in the literature (Section 1.3.1), the research challenges we face in our work (Section 1.3.2). Sections 1.3.3 and 1.3.4 present the scope and the contributions of this research.

1.3.1 Problems in the Literature

- **Problem 1**: As shown by Figure 1, the proportion of empirical studies and studies based on mining software repositories regarding to software quality has been increasing exponentially since 2010 ([KZWG11, LNH+11, SLKJ11, BN11, TSL12, ZNGM12, SNH13, CNSH14, MANH14, HNH15] are some noticeable examples). While hundreds of bug prediction papers have been published by academia over the last decade, the developed tools and approaches fail to change developer behavior while deployed in industrial environment [LLS+13]. This is mainly due to the lack of actionable messages, i.e. messages that provide concrete steps to resolve the problem at hand.

- **Problem 2**: The literature contains numerous papers about tools that improve the overall software quality with static [Dan00, Bur03, Hov07, MGDL10] and dynamic [NPMG12, NMV13, Pal13] analysis. To the best of our knowledge approaches leveraging other sources to improve quality or efficiency mostly rely on web-search [BGL+09, RYR13, MBFV13].

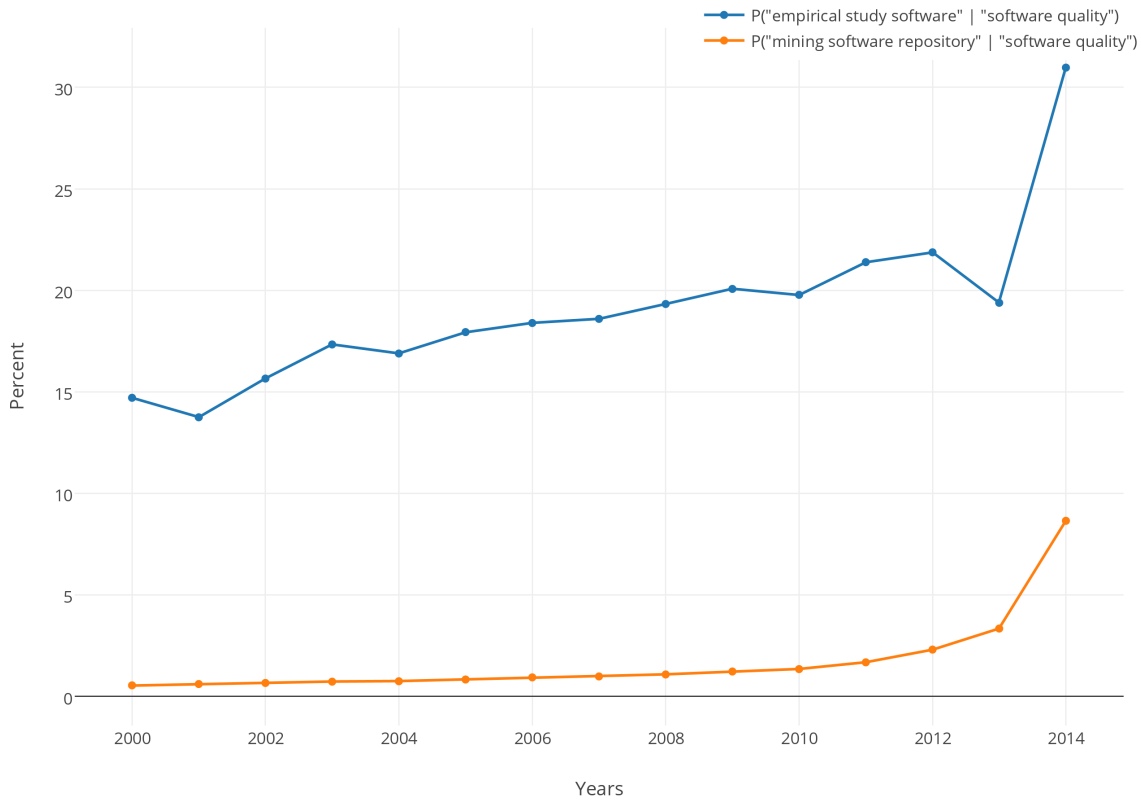

Figure 1: Proportion of papers containing "Empirical Study" or "Mining software repository" with regards to the paper in Software quality indexed by Google Scholar

- **Problem 3**: There is no approach that supports the natural language search and comparison of issues, source code and tasks regardless of the project, repository, revision and issue management system and programming language. Such an approach could dramatically transform software engineering processes. Moreover, the data contained in these repositories lack a taxonomy, as for clone detection [Cor], to classify the research.

1.3.2 Research Challenges

- **Challenge 1** : Issues & projects and revision systems are plenty and they all have specific processes and limitations. Mining them all in order to have a representative model is challenging. Despite the parsing aspect, terabytes of new data are generated every day thus, storing and accessing to these data in reasonable time will require

innovations in high density nosql databases [Nay14] and web servers [Nay13b]. Moreover, creating the relationship between both systems is still an open issue as discussed in sections 1.1.2 and 2.2.

- **Challenge 2**: Providing code samples broke down to the right level, at the right time in order to solve a problem or improve the current code in terms of quality, performances or reliability during a programming session will force us to improve current approaches of source code transformation and normalization [Cor06a, Cor06b, RC08, CR11].

1.3.3 Scope of the research

The area of this research is to improve the processes of software engineering by providing contextual information, in order to improve the quality, the performance, the reliability of a given code during a programming session. These contextual information will come from the mining issue & project and revision systems. Hence, we will not define what are the good or bad practices to improve the quality, the performance, the reliability of a given code but rely on the mined data.

1.3.4 Thesis contributions

Figure 2 depicts our proposed solution for fulfilling our objectives. First of all, an end-user (team member) will report an issue (open a task) in one the organization issues & project management system. This can be done in *Sourceforge*, *Bugzilla*, *JIRA* or *Github* which are the systems we want to support first as described in section 1.1.2.
Issues (tasks) are mapped with their fixes (implementations) inside `BUMPER` (BUg MetarePository for dEvelopers and Researchers). The source code is fetched from the supported version system: *Git*, *Svn*, *Mercurial* presented in section 1.1.1. `BUMPER` is a meta-repository that makes issues, tasks and related source code searchable using natural language (in opposition to structured query language). When an issue is reported, `JCHARMING` (Java CrasH Automatic Reproduction by directed Model checkING) will fetch the content of the issue and try to create a scenario to reproduce the on-field crash. In case of success, the developer assigned to this issue will be notified and the scenario stored in `BUMPER`. The developer assigned to the task or the issue will modify the code and, in real time, very much like intellisense or auto-completion in modern IDE, `RESEMBLE` (REcommendation System based on cochangE Mining at Block LEvel) will propose improvements or follow up on the developer's code using the decades of history stored in `BUMPER`. Once s/he is done, s/he submit a commit, patch or diff to the version system. However, before s/he allowed to do so, `BIANCA` (Bug

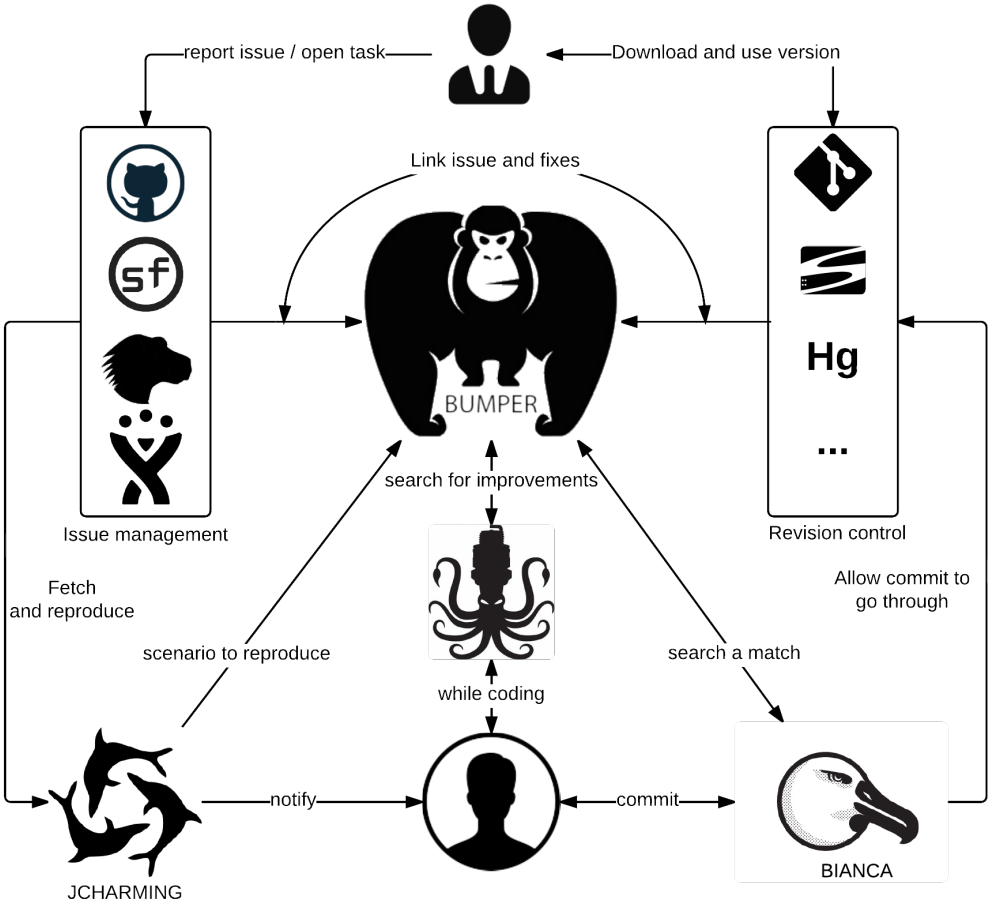

Figure 2: Proposed Architecture

Insertion ANticipation by Clone Analysis at commit time) will kick in and query `BUMPER` looking for similar modifications in other projects and even other programming languages that led to the insertion of a defect in order to warn the user about potential hazardous code.

The bug taxonomy required to build `BUMPER`, `BUMPER` itself, `JCHARMING`, `RESSEMBLE` and `BIANCA` are presented in sections 3.1, 3.2, 3.3, 3.4 and 3.5, respectively. In addition, we list parts that have been published in peer-reviewed conferences in section 4.2 and our publication plan in 4.4.

As a motivating example, we draft the following scenario. Table 1 presents hypothetical data stored in `BUMPER` in terms of sequence #id, sequence of code blocks, a flag to know if a said sequence introduced an issue in a given system and step to reproduce the issue if any.

Seq #ID	Language #ID	Blocks	Root of Issue	Steps to reproduce
1	1	A-A-B-C-A-A	Yes	E-F-G
2	1	A-A-B-C	No	-
3	2	D-E-A-C	No	-

Table 1: Hypothetical `BUMPER` data

During a programming session, let's assume that a developer has implemented $A - B - C$, then `RESEMBLE` will recommends to transform the current code to $A - A - B - C$ as it seems to be the right thing to do. If the developer follows `RESEMBLE` recommendation and then adds another two As, the sequence is now $A - A - B - C - A - A$. If the developer commit its changes, `BIANCA` will raise a warning saying that this sequence is known to be the root of an issue and invite the developer to execute the steps $E - F - G$ – that were produced by `JCHARMING` – in order to see if s/he did introduce a defect. Moreover, `BIANCA` will take the time to compare $A - A - B - C - A - A$ and $D - E - A - C$ using our normalization algorithms even if they are not in the same programming language. Finally, when a new issue is submitted, `BUMPER` indexes it and `JCHARMING` tries to reproduce it and update the step to reproduce part of `BUMPER`.
We can envision the potential of such a system and its complexity, knowing that it would contain millions of issues, hundreds of thousand projects, dozens of programming languages and will help developers leveraging the knowledge of other developers.
To summarize this thesis have four main contributions:

- To provide a taxonomy of software issues to classify the research.
- To propose approaches to aggregate as many issues and revisions systems as possible.
- To propose approaches to reproduce field crashes in a lab environment using the issue content.
- To propose a context-aware IDE that will improve day-to-day programming session with concise and appropriate code samples.

1.4 Outline

The remainder of this proposal is organized as follows. Chapter 2 summarizes the related work. Chapter 3 introduces the proposed approache and the preliminary experiments along with the results. Chapter 4 presents the current state of the research, and highlights our

key current contributions and future research directions along with a possible publication schedule. Finally, Chapter 5 provides some concluding remarks and future works.

Chapter 2

Related Work

2.1 Crash reproduction

In his Ph.D thesis [Che13], Chen proposed an approach named STAR (Stack Trace based Automatic crash Reproduction). Using only the crash stack, STAR starts from the crash point and goes backward towards the entry point of the program. During the backward process, STAR computes the required condition to reach the crash point using an SMT (Satisfiability Modulo Theories) solver named Yices [DM06]. 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. When applied to different systems, STAR achieved 60% accuracy.

Jaygarl et al. [JKXC10] created OCAT (Object Capture based Automated Testing). The authors' approach starts by capturing objects created by the program when it runs on-field in order to provide them in an automated test process. Indeed the coverage of automated tests is often low due to the 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 once more. While not targeting the reproduction of a bug, OCAT is a well-known approach and was used as the main mechanism for bug reproduction.

Narayanasamy et al. [NPC05] 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, however, the instrumentation of the system is problematic since it slows down considerably the execution.

Jin et al. [JO12] proposed BugRedux for reproducing field failures for in-house debugging. 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 stacks, and points of failure. The data that successfully reproduced the field crash is sent to software developers to fix the bug.

Based on the success of BugRedux, the authors built F3 (Fault localization for Field Failures) [JO13]. F3 performs many executions of a program on top of BugRedux in order to cover different paths 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.

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% of bug reproduction can only be obtained through instrumentation and with an overhead ranging from 1% to 1066%.

Clause et al. [CO07] proposed a technique for enabling and supporting debugging of field failures. They record the execution of the program on the client side and propose to compress the generated data to the minimal required size to ensure that the reproduction is feasible. This compression is also performed on the client side. Moreover, the authors keep traces of all accessed documents in the operating system and also compress/reduce them to the minimal. Overall, they are able to reproduce on-field bug using a file weighting 70Kb. The minimal execution paths triggering the failure are then sent to the developers who can replay the execution on a sandbox, simulating the clients environment. While efficient, this approach suffers from severe security and privacy issues.

RECORE (REconstructing CORE dumps) is a tool proposed by Rossler et al. [RZF+13]. It instruments Java bytecode to wrap every method in a try and catch block while keeping a quasi-null overhead. The tool starts from the core dump and tries (with evolutionary

algorithms) to reproduce the same dump by executing the programs many times. The set of inputs responsible for the failure is generated when the generated dump matches the collected one. ReCrash [AKE08] is a tool that aims to make software failures reproducible by preserving object states. It uses 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 on the buggy code. This approach suffers from overhead when they record everything (between 13% to 64% in some cases). The authors also propose an alternative 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 when (and if) it appears.

JRapture [SCFP00] is a capture/replay tool for observation-based testing. The tool captures execution of Java programs to replay it in-house. To capture the execution of a Java program, the authors used their own version of the Java Virtual Machine (JVM) and employ a lightweight, transparent capture process. Using their own JVM allows one to capture any interactions between a Java program and the system, including GUI, file, and console inputs, and on replay, it presents each thread with exactly the same input sequence it saw during capture. Unfortunately, they have to make their customer use their own JVM in order to support their approach, which limits the generalization of the approach to mass-market software.

Finally, Zamfir et al. [PO11] proposed ESD, an execution synthesis approach which 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.

Except for STAR, approaches targeting 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, instrumentation can cause a massive overhead (1% to 1066%) while running the system. In addition, data generated at run-time using instrumentation may contain sensitive information.

2.2 Issue and source code relationships

Researchers started studying the relationships between issues and source code repositories more than two decades ago. To the best of our knowledge the first ones who conduct this type of study on a significant scale were Perry and Stieg [PS93]. In these two decades, many aspects of these relationships have been studied in length. For example, researchers interested themselves in ameliorating the issues report by specifying guidelines to make a good report [BJS+08] and try to further simplify the existing models [HGGBR08].
Then, we can find approaches on how long it will take for an issue to get fixed [BN11, ZGV13, SKP14] and where it should be fixed [ZZL12, KTM+13]. With the rapidly increasing number of issues, the community also interested itself in prioritizing the issues report compared to one another [KWM+11] and do so by predicting the severity of an issue [LDGG10].
Finally, researchers proposed approaches to predict which issues will get reopened [ZNGM12, Lo13] which issues report is a duplicate of which other one [JW08, BPZ08, TSL12].
Another field of study consists in assigning these issues reports, if possible automatically to the right developers through triaging [AHM06, JKZ09, TNAKN11, BvdH13] and predicting which locations are likely to yield new bugs [KZPJ06, KZWZ07a].

2.3 Crash Prediction

Predicting crash, fault and bug is very large and popular research area. The main goal behind the plethora of papers is to save on manpower—being the most expensive resource to build software—by directing their efforts on locations likely to contain a bug, fault or crash.
There are two distinct trends in crash, fault and bug prediction in the papers accepted to major venues such as MSR, ICSE, ICSME and ASE: history analysis and current version analysis.
In the history analysis, researchers extract and interpret information from the system. The idea being that the files or locations that are the most frequently changed are more likely to contain a bug. Additionally, some of these approaches also assume that locations linked to a previous bug are likely to be linked to a bug in the future.
On the other hand, approaches using only the current version to predict bugs assume that the current version, i.e. its design, call graph, quality metrics and more, will trigger the appearance of the bug in the future. Consequently, they do no require the history and only need the current source-code.
In the remaining of this section, we will describe approaches belonging to the two families.

Change logs approaches

Change logs based approaches rely on mining the historical data of the application and more particularly, the source code *diffs*. A source code *diffs* contains two versions of the same code in one file. Indeed, it contains the lines of code that have been deleted and the one that has been added. It is worth noting that, *diffs* files do not represent the concept of modified line. Indeed, a modified line will be represented by a deletion and an addition. Researchers mainly use five metrics when dealing with *diffs* files:

- Number of files: The number of modified files in a given commit
- Insertions: The number of added lines
- Deletions: The number of deleted lines
- Churns: The number of deleted lines immediately followed by an insertion which give an approximation of how many lines have been modified
- Hunks: The number of consecutive blocks of lines. This gives an approximation of how many distinct locations have been edited to accomplish a unit of work.

Naggapan *et al.* studied the churns metric and how it can be connected to the apparition of new defect in a complex software systems. They established that relative churns are, in fact, a better metric than classical churn [NB05a] while studying Windows Server 2003.
Hassan, interested himself with the entropy of change, i.e. how complex the change is [Has09]. Then, the complexity of the change, or entropy, can be used to predict bugs. The more complex a change is, the more likely it is to bring the defect with it. Hassan used its entropy metric, with success, on six different systems. Prior to this work, Hassan, in collaboration with Holt proposed an approach that highlights the top ten most susceptible locations to have a bug using heuristics based on *diffs* file metrics [HH05]. Moreover, their heuristics also leverage the data of the bug tracking system. Indeed, they use the past defect location to predict new ones. The conclusion of these two approaches has been that recently modified and fixed locations where the most defect-prone compared to frequently modified ones.
Similarly to Hassan and Hold, Ostrand *et al.* predict future crash location by combining the data from changed and past defect locations [OWB05]. The main difference between Hassan and Hold and Ostrand *et al.* is that Ostrand *et al.* validate their approach on industrial systems as they are members of the AT&T lab while Hassan and Hold validated their approach on open-source systems. This proved that these metrics are relevant for open-source and industrial systems.

Kim *et al.* applied the same recipe and mined recent changes and defects with their approach named bug cache [KZWZ07b]. However, they are more accurate than the previous approaches at detecting defect location by taking into account that is more likely for a developer to make a change that introduces a defect when being under pressure. Such changes can be pushed to revision-control system when deadlines and releases date are approaching.

Single-version approaches

Approaches belonging to the single-version family will only consider the current version of the software at hand. Simply put, they don't leverage the history of changes or bug reports. Despite this fact, that one can see as a disadvantage compared to approaches that do leverage history, these approaches yield interesting results using code-based metrics.
Chidamber and Kemerer published the well-known CK metrics suite [CK94] for object oriented designs and inspired Moha *et al.* to publish similar metrics for service oriented programs [MPN+12]. Another famous metric suite for assessing the quality of a given software design is Briand's coupling metrics [BDW99].
The CK and Briand's metrics suites have been used, for example, by Basili *et al.* [BBM96], El Emam *et al.* [EMM01], Subramanyam *et al.* [SK03] and Gyimothy *et al.* [GFS05] 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.* [NPMG12, Nay13a], Demange *et al.* [DMT13] and Palma *et al.* [Pal13] used Moha et *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.
Finally, Nagappan *et al.* [NB05b, NBZ06] and Zimmerman [ZPZ07, ZN08] further refined metrics-based detection by using statical analysis and call-graph analysis.
While hundreds of bug prediction papers have been published by academia over the last decade, the developed tools and approaches fail to change developer behavior while deployed in industrial environment [LLS+13]. This is mainly due to the lack of actionable message, i.e. messages that provide concrete steps to resolve the problem at hand.
`pErICOPE` (Ecosystem Improve source COde during Programming session with real-time mining of common knowlEdge), our proposed ecosystem, will be different in a sense that we will provide actionable messages as presented in Section 1.3.4.

Chapter 3

Methodology

In this chapter, we present, in the details, each component involved in our proposed solution (Section 1.3.4). First, we describe the bug taxonomy we created in section 3.1, then we present our four different approaches `BUMPER`, `JCHARMING`, `RESEMBLE` and `BIANCA` in sections 3.2, 3.3, 3.4 and 3.5, respectively.

3.1 Bug Taxonomy

We can reason about types of bugs at different levels. For example, we can group bugs based on the developers that fix them or using information about the bugs such as crash traces. In this paper, we are interested in bugs that share similar fixes. By a fix, we mean a modification (adding or deleting lines of code) to an exiting file that is used to solve the bug. With this in mind, the relationship between bugs and fixes can be modeled using the UML diagram in Figure 3. The diagram only includes bugs that are fixed. 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 4).

Figure 3: Class diagram showing the relationship between bugs and fixed

The first and second types are the ones we intuitively know about. Type 1 refers to a bug being fixed in one single location (i.e., one file), while Type 2 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. Type 3 refers to multiple bugs that are fixed in the exact same location. Type 4 is an extension of Type 3, where

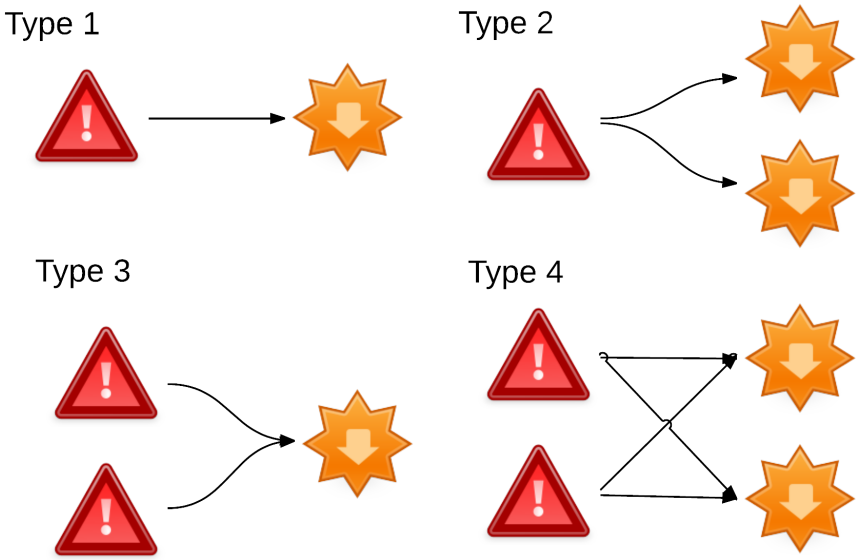

Figure 4: Proposed Taxonomy of Bugs

multiple bugs are resolved by modifying the same set of locations. Note that Type 3 and Type 4 bugs are not duplicates, they may occur when different features of the system fail due to the same root causes (faults). We conjecture that knowing the proportions of each type of bugs in a system may provide insight into the quality of the system. Knowing, for example, that in a given system the proportion of Type 2 and 4 bugs is high may be an indication of poor system quality since many fixes are needed to address these bugs. In addition, the existence of a high number of Types 3 and 4 bugs calls for techniques that can effectively find bug reports related to an incoming bug during triaging. This is similar to the many studies that exist on detection of duplicates (e.g., [RAN07, SLW+10, NNN+12]), except that we are not looking for duplicates but for related bugs (bugs that are due to failures of different features of the system, caused by the same faults). To our knowledge, there is no study that exmpirically examines bug data with these types in mind, which is the main objective of this paper. More particularly, we are interested in the following research questions:

- RQ1: What are the proportions of different types of bugs?
- RQ2: How complex is each type of bugs?
- RQ3: How fast are these types of bugs fixed?

3.1.1 Study Setup

Figure 5 illustrates our data collection and analysis process that we present here and discuss in more detail in the following subsections. First, we extract the raw data from the two bug report management systems used in this study (Bugzilla and Jira). Second, we extract the fix to the bugs from the source code version control system of Netbeans and Apache (Maven and Git).

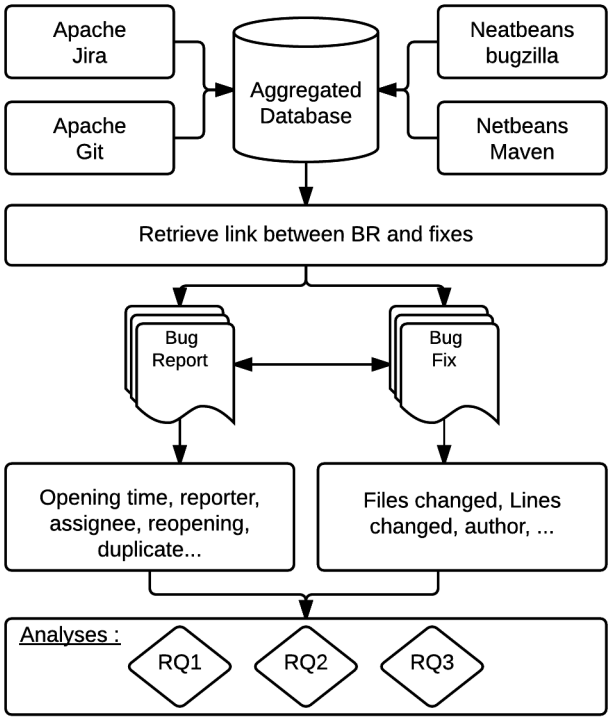

Figure 5: Data collection and analysis process of the study

The extracted data is consolidated in one database where we associate each bug report to its fix. We mine relevant characteristics of BRs and their fixes such as opening time, number of comments, number of times the BR is reopened, number of changesets for BR and the number of files changed and lines modified for fixes or patch. Finally, we analyze these characteristics to answer the aforementioned research questions (RQ).

3.1.2 Datasets

In this study, we used two distinct datasets: Netbeans and the Apache Software Foundation projects. Netbeans is an integrated development environment (IDE) for developing with many languages including Java, PHP, and C/C++. The very first version of Netbeans,

then known as Xelfi, appeared in 1996. The Apache Software Foundation is a U.S non-profit organization supporting Apache software projects such as the popular Apache web server since 1999. The characteristics of the Netbeans and Apache Software Foundation datasets used in this paper are presented in Table 2.

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 2: Datasets

Cumulatively, these datasets span from 2001 to 2014. In summary, our consolidated dataset contains 102,707 bugs, 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 the bugs, identified 3,284 sub-projects, and 17,984 unique contributors to these bug report and source code version management systems. Finally, the cumulated opening time for all the bugs reaches 10,661 working years (3,891,618 working days).

3.1.3 Study Design

We describe the design of our study by first stating the research questions, and then explaining the variables, and analysis methods we used to answer these questions. We formulate three research questions (RQs) with the ultimate goal to improve our understanding of each bug type. We focus, however, on Types 2 and 4. This is because these bugs require multiple fixes. They are therefore expected to be more complex. The objective of the first research question is to analyze the proportion of each type of bugs. The remaining two questions address the complexity of the bugs and the bug fixing duration according to the type of bugs. 1)

RQ 1: What are the proportions of different types of bugs?

The answer to this question provides insight into the distribution of bugs according to their type with a focus on Type 2 and 4 bugs. As discussed ealier, knowing, for example, that bugs of Type 2 and 4 are the most predominant ones suggests that we need to investigate techniques to help detect whether an incoming bug is of Types 2 and 4 by examining historical data. Similarly, if we can automatically identify a bug that is related to another one

that has been fixed then we can reuse the results of reproducing the first bug in reproducing the second one.

Hypothesis: To answer this question, we analyze whether Type 2 and 4 bugs are predominant in the studied systems, by testing the null hypothesis:

- H_{01A} : The proportion of Types 2 and 4 does not change significantly across the studied systems

We test this hypothesis by observing both a global (across systems) and a local predominance (per system) of the different types of bugs. We must observe these two aspects to ensure that the predominance of a particular type of bug is not circumstantial (in few given systems only) but is also not due to some other, unknown factors (in all systems but not in a particular system).

Variables: We use as variables the amount of resolved/fixed bugs of each type (1, 2, 3 and 4) that are linked to a fix (commit). As mentioned earlier, duplicate bugs are excluded. These are marked as resolved/duplicate in our dataset.

Analysis Method: We answer RQ1 in two steps. The first step is to use descriptive statistics; we compute the ratio of Types 2 and 4 bugs and the ratio of Types 1 and 3 bugs to the total number of bugs in the dataset. This shows the importance of Types 2 and 4 bugs compared to Types 1 and 3 bugs.

In the second step, we compare the proportions of the different types of bugs with respect to the system where the bugs were found. We build the contingency table with these two qualitative variables (the type and studied system) and test the null hypothesis H 01A to assess whether the proportion of a particular type of bugs is related to a specific system or not.

We use the Pearson's chi-squared test to reject the null hypothesis H_{01A} . Pearsons chi-squared independence test is used to analyze the relationship between two qualitative data, in our study the type bugs and the studied system. The results of Pearsons chi-squared independence test are considered statistically significant at $\alpha = 0.05$. If p-value ≤ 0.05, we reject the null hypothesis H_{01A} and conclude that the proportion of types 3 and 4 bugs is different from the proportion of type 1 and 2 bugs for each system.

RQ 2: How complex is each type of bugs?

We address the relation between Types 2 and 4 bugs and the complexity of the bugs in terms of severity, duplicate and reopened.We analyze whether Types 2 and 4 bugs are more complex to handle than Types 1 and 3 bugs, by testing the null hypotheses:

- H_{02S} : The severity of Types 2 and 4 bugs is not significantly different from the severity of Types 1 and 3
- H_{02D} : Types 2 and 4 bugs are not significantly more likely to get duplicated than Types 1 and 3.
- H_{02R} : Type 2 and 4 bugs are not significantly more likely to get reopened than Types 1 and 3.

Variables: We use as independent variables for the hypotheses H_{02S} , H_{02D} , H_{02R} the bug type (if the bug is from Types 2 and 4 or if it is from Types 1 and 3). For H_{02S} we use the severity as dependent variable to assess the relationship between the bug severity and the bug type. For H_{02D} (respectively H_{02R}) we use a dummy variable duplicated (reopened) to assess if a bug has been duplicated (reopened) at least once or not. This will be used to assess the relationship between the type of the bugs and the fact that the bug is more likely to be reopened or duplicated.
Analysis Method: For each hypothesis, we build a contingency table with the qualitative variables type of bugs (2 and 4 or 1and 3) and the dependent variable duplicated (respectively reopened) and the severity variable.
We use the Pearsons chi-squared test to reject the null hypothesis H_{02D} (respectively H_{02R}) and H_{02S}. The results of Pearsons chi-squared independence test are considered statistically significant at $\alpha = 0.05$. If a p-value ≤ 0.05, we reject the null hypothesis H_{02D} (respectively H_{02R}) and conclude the fact that the bug is more likely to be duplicated (respectively reopened) is related to the type of the bug and we reject H_{02S} and conclude that the severity level of the bug is related to the bug type.

RQ 3 : How fast are these types of bugs fixed ?

In this question, we study the relation between the different types of bugs and the fixing time. We are interested in evaluating whether developers take more time to fix Types 2 and 4 bugs than Type 1 and 3, by testing the null hypothesis:

- H_{03} : There is no statistically-significant difference between the duration of fixing periods for Types 2 and 4 bugs and that of Types 1 and 3 bugs.

Variables: To compare the bug fixing time with respect to their type, we use as independent variable the type Ti of a bug Bi, to distinguish between Types 1 and 3 bugs and Types 2 and 4 bugs. We consider as dependent variable the fixing time, FTi, of the bug Bi. We

compute the fixing time FTi of a bug Bi. The fixing time FTi is the time between when the bug is submitted to when it is closed/fixed.

Analysis Method: We compute the (non-parametric) Mann- Whitney test to compare the BR fixing time with respect to the BR type and analyze whether the difference in the average fixing time is statistically significant. We use the Mann- Whitney test because, as a non-parametric test, it does not make any assumption on the underlying distributions. We analyze the results of the test to assess the null hypothesis H_{03} . The result is considered as statistically significant at $\alpha = 0.05$. Therefore, if p-value ≤ 0.05, we reject the null hypothesis H 03 and conclude that the average fixing time of Types 1 and 3 bugs is significantly different from the average fixing time of Types 2 and 4 bugs.

3.1.4 Study result and discussion

In this section, we report on the results of the analyses we performed to answer our research questions. We then dedicate a section to discussing the results.

RQ 1 : What are the proportions of different types of bugs?

Figure 6 shows the percentage of the different types of bugs. As shown in the figure, we found that 65% of the bugs are from Types 2 and 4. This shows the predominance of this type of bugs in all the studied systems. Figure 5 shows the repartition per dataset. We can see that Netbeans and Apache have 66% and 64% bugs of Type 1and 3, respectively. To ensure that this observation is not related to a particular system, we perform Pearsons chi-squared test across the studied systems. Table 3 shows the contingency table for the studied systems and the result of Pearsons chi-squared test. The results show that there is statistically significant difference between the proportions of the different types of bugs.

System	Type 1 and 3	Type 2 and 4	Pearsons chisquared p-value
Apache	4910	8626	p-value <0,0001
Netbeans	9050	17586	

Table 3: Contingency table and Pearson's chi-squared tests

Table 4 shows the number of bugs for each type of bugs and the percentage of each type of bugs. We can see that Types 3 and 4 bugs represent 28.33% and 61.21% of the total of bugs, respectively. Types 1 and 2 represent only 6.78% and 3.74%. Together, Types 3 and 4 bugs represent almost 90% of the total number of bugs linked to a commit.

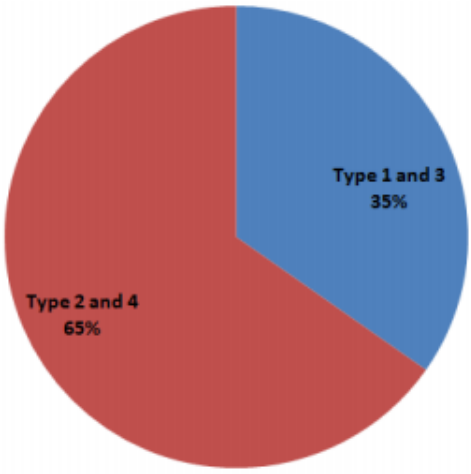

Figure 6: Proportions of different types of bugs

Datasets	T1	T2	T3	T4	Total
Netbeans	776 (2.90%)	240 (0.90%)	8372 (31.29%)	17366 (64.91%)	26754
Apache	1968 (14.32%)	1248 (9.08%)	3101 (22.57%)	7422 (54.02%)	13739
Total	2744 (6.78%)	1488 (3.74%)	11473 (28.33%)	24788 (61.21%)	40493

Table 4: Proportion of bug types in amount and percentage

We can thus reject the null hypothesis H_{01A} and conclude that there is a predominance of Types 2 and 4 bugs in all studied systems and this observation is not related to a specific system.

RQ 2 : How complex is each type of bugs?

Figure 7 and 8 show the proportion of each bug type with respect to their severity for each dataset. Table V shows the proportion of each bug type with repect to their severity and dataset. For Netbeans, the bugs we examined in our dataset are either labeled as Blocker or Normal (despite the fact that Netbeans uses Bugzilla that supports all the severity levels presented in the previous section).

For the Apache dataset, the severity levels range from Blocker to Trivial as shown in Figure 7. Figure 8 shows that in Netbeans around 67% of Types 2 and 4 bugs are normal. The same holds for Types 1 and 3 bugs (66% are considered of normal severity). This indicates

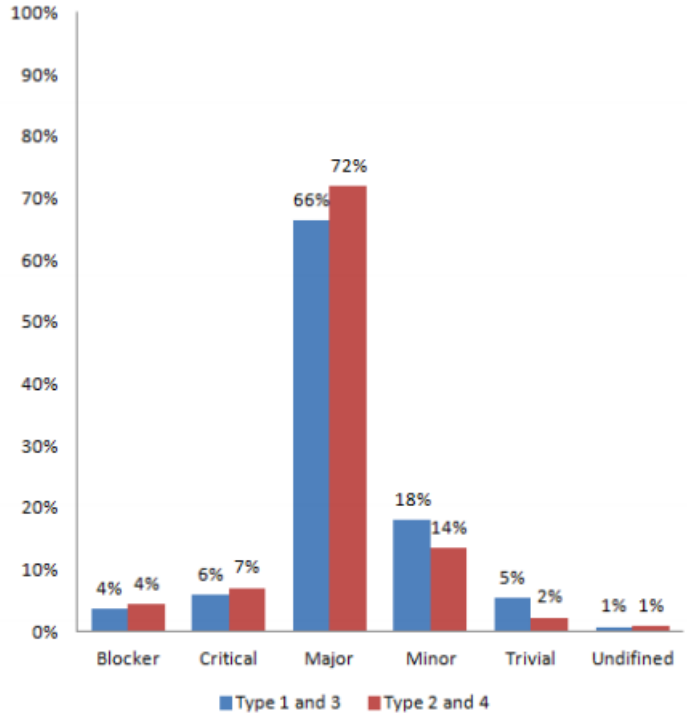

Figure 7: Proportions of Types 1 and 3 versus Types 2 and 4 with respct to their severity in the Apache dataset.

that most Types 2 and 4 bugs and Types 1 and 3 bugs are not critical in the Netbeans dataset. For the Apache dataset, the results indicate that the majority of the bugs are considered of major severity (66% for Types 1 and 3 and 72% for Types 2 and 4). It is challenging to understand the discrepancy between the two datasets partly because of the way the severity is assigned to BRs.

Table 5 shows the result of the Pearson chi-squared tests for the H_{02S}, H_{02D} and H_{02R} hypotheses.

According to the results of the test (p-value ¡ 0.005), we reject the null hypothesis H_{02S} and conclude that there is a significant difference between the severity of Types 1 and 3 bugs and the severity of Types 2 and 4 bugs.

Table 7 shows the occurrences of duplicate and reopened bugs with respect to their bug type in each dataset. In Netbeans, the proportion of Type 1 bugs that are marked as source of duplicate is 6.06%, 4.59% for Type 2 bugs, 5.09% for Type 3 bugs and 5.87% for Type 4 bugs with a total of 1503 bugs over 26754 (5.62%). In Apache, the proportion of Type 1 bug marked a source of a duplicate is 2.59% and 2.24%, 1.61% and 2.91% for Types 2, 3 and 4, respectively.

Second, we analyze the reopened bugs to see the link between the reopening and the type

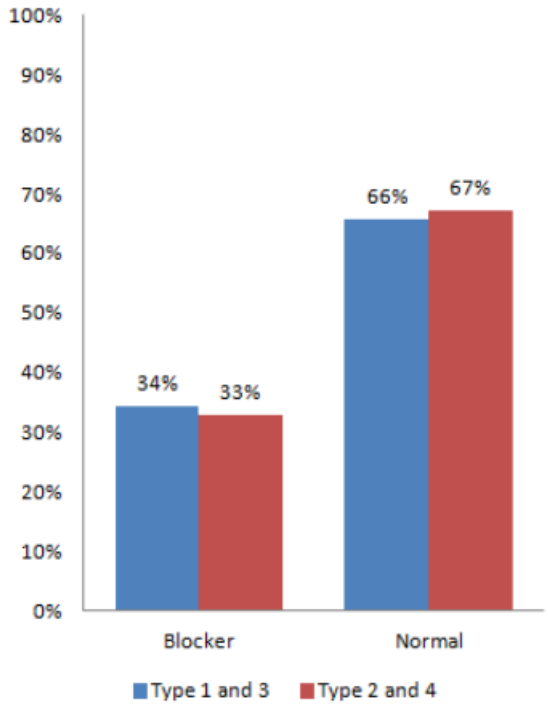

Figure 8: Proportions of Types 1 and 3 versus Types 2 and 4 with respct to their severity in the Netbeans dataset.

System	Factor	Pearsons chisquared p-value
Apache	Severity	p-value <0.005
	Reopened	p-value <0.005
	Duplicated	p-value <0.005
Netbeans	Severity	p-value <0.005
	Reopened	p-value <0.005
	Duplicated	p-value <0.005

Table 5: Pearson's chi squared p-values for the severity, the reopen and the duplicate factor with respect to a dataset

of bugs. We perform Pearsons chi-squared test to reject the null hypothesis H_{02R}.

According to the results of the test (p-value ¡ 0.005), we reject the null hypothesis H_{02R} and conclude that there is a significant relationship between the reopening of a bug and its type.

Third, we analyze the duplicated bugs to see if there is a link between the bug type and the fact duplication. We perform Pearsons chi-squared test to reject the null hypothesis H_{02D}.

Severity	T1	T2	T3	T4
Netbeans				
Blocker	340 43.81%	109 45.42%	2850 34.04%	5687 32.75%
Normal	436 56.19%	131 54.58%	5522 65.96%	11678 67.25%
Total	776 100%	240 100%	8372 100%	17365 100%
Apache				
Blocker	68 3.46%	53 4.25%	115 3.71%	329 4.43%
Critical	84 4.27%	44 3.53%	213 6.87%	565 7.61%
Major	1245 63.26%	811 64.98%	2096 67.59%	5427 73.12%
Minor	408 20.73%	276 22.12%	501 16.16%	899 12.11%
Trivial	113 5.74%	31 2.48%	159 5.13%	161 2.17%
Total	1918 100%	1215 100%	3084 100%	7381 100%

Table 6: Proportion of each bug type with respect to severity.

According to the results of the test (p-value ¡ 0.005), we reject the null hypothesis H_{02D} and conclude that there is a significant relationship between the duplication of a bug and its type.

RQ 3 : How fast are these types of bugs fixed ?

Figure 9 shows the fixing time for Types 1 and 3 versus Types 2 and 4 for Netbeans and the Apache Software Foundation. In Netbeans, 98.96 and 137.05 days are required to fix Types 1 and 3 and Types 2 and 4, respectively. In Apache, 55.76 and 85.48 days are required to fix Types 1 and 3 and Types 2 and 4, respectively.

Table 8 shows the average fixing time of bugs with respect to their bug type in each dataset. We analyze the difference in the fixing time of bugs with respect to their bug type by conducting a Mann-Whitney test to assess $H03$.The results show that the difference between

Type	T1	T2	T3	T4	Total
Netbeans					
Dup.	6.06% (47)	4.59% (11)	5.09% (426)	5.87% (1019)	5.62% (1503)
Reop.	4.38% (34)	7.08% (17)	4.81% (403)	7.09% (1231)	6.30% (1685)
Apache					
Dup	2.59% (51)	2.24% (28)	1.61% (50)	2.91% (216)	2.51% (345)
Reop	5.59% (110)	6.49% (81)	3.10% (96)	6.90% (512)	5.82% (799)
Total					
Dup	3.57% (98)	2.62% (39)	4.15% (476)	4.98% (1235)	4.56% (1848)
Reop	5.25% (144)	6.59% (98)	4.35% (499)	7.03% (1743)	6.13% (2484)

Table 7: Percentage and occurences of bugs duplicated by other bufs and reopenned with respect to their bug type and dataset.

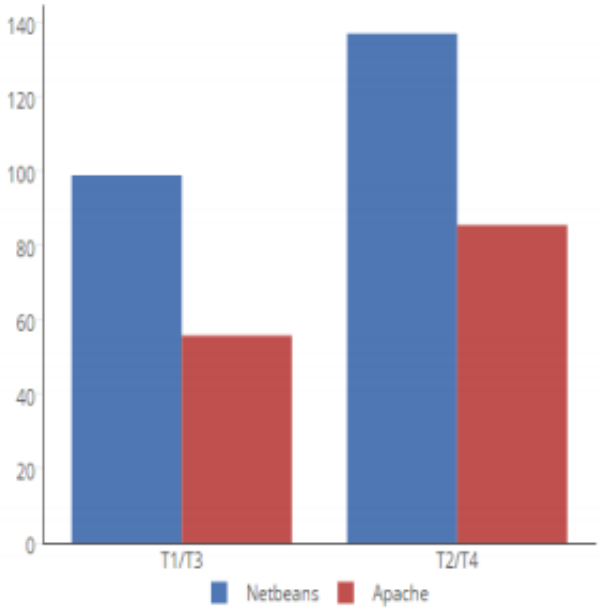

Figure 9: Fixing time of Types 1 and 3 versus fixing time of Types 2 and 4.

Dataset	T1	T2	T3	T4	Average
Netbeans	97.66	117.42	100.26	156.67	118.00
Apache	73.48	118.12	38.04	52.83	70.62
Total	85.57	117.77	69.15	104.75	94.31

Table 8: Average fixing time with respect to bug type

the fixing time of Types 2 and 4 and Types 1 and 3 is statistically significant (p-value ¡ 0,005).

Therefore, we can reject the null hypothesis $H03$ and conclude that the fixing of Types 2 and 4 bugs takes more time than the fixing of Types 1 and 3 bugs.

Dicussion

Repartition of bug types: One important finding of this study is that there is significantly more Types 2 and 4 bugs than Types 1 and 3 in all studied systems. Moreover, this observation is not system-specific. The traditional one-bug/ one-fault way of thinking about bugs only accounts for 35% of the bugs. We believe that, recent triaging algorithms [JW08, JKZ09, KCZH11, TNAKN11] can benefit from these findings by developing techniques that can detect Type 2 and 4 bugs. This would result in better performance in terms of reducing the cost, time and efforts required by the developers in the bug fixing process.

Severity of bugs: We discussed the severity and the complexity of a bug in terms of its likelihood to be reopened or marked as duplicate (RQ2). Although clear guidelines exist on how to assign the severity of a bug, it remains a manual process done by the bug reporter. In addition, previous studies, notably those by Khomh et al. [KCZH11], showed that severity is not a consistent/trustworthy characteristic of a BR, which lead to he emergence of studies for predicting the severity of bugs (e.g., [LDGG10, LDSV11, TLS12]). Nevertheless, we discovered that there is a significant difference between the severities of Types 1 and 3 compared to Types 2 and 4.

Complexity of bugs: At the complexity level, we use the number of times a bug is reopened as a measure of complexity. Indeed, if a developer is confident enough in his/her fix to close the bug and that the bug gets reopened it means that the developer missed some dependencies of the said bug or did not foresee the consequences of the fix. We found that there is a significant relationship between the number of reopenings and type of a bug. In other words, there is a significant relationship between the complexity and the type of a given bug. In our datasets, Types 1 and 3 bugs are reopened in 1.88% of the cases, while Types 2 and 4 are reopened in 5.73%. Assuming that the reopening is a representative

metric for the complexity of bug, Types 2 and 4 are three times more complex than Types 1 and 3. Finally, if we consider multiple reopenings, Types 2 and 4 account for almost 80% of the bugs that reopened more than once and more than 96% of the bug opened more than twice. While current approaches aiming to predict which bug will be reopen use the amount of modified files [SIK+10, ZNGM12, Lo13], we believe that they can be improved by taking into account the type of a the bug. For example, if we can detect that an incoming bug if of Type 2 or 4 then it is more likely to reopened than a bug of Type 1 or 3. Similarly, approaches aiming to predict the files in which a given bug should be fixed could be categorized and improved by knowking the bug type in advance [ZZL12, KTM+13].

Impact of a bug: To measure the impact of bugs in end-users and developers, we use the number of times a bug is duplicated. This is because if a bug has many duplicates, it means that a large number of users have experienced and a large number of developers are blocked the failure. We found that there is a significant relationship between the bug type and the fact that it gets duplicated. Types 1 and 3 bugs are duplicated in 1.41% of the cases while Types 2 and 4 are duplicated in 3.14%. Assuming that the amount of duplication is an accurate metric for the impact of bug, Types 2 and 4 have more than two times bigger impact than Types 1 and 3. Similarly to reopening, if we consider multiple duplication, Types 2 and 4 account for 75% of the bugs that get duplicated more than once and more than 80% of the bugs that get duplicated more than twice. We believe that approaches targeting the identification of duplicates [BPZ08, JW08, SLW+10, TSL12] could leverage this taxonomy to achieve even better performances in terms of recall and precision.

Fixing time: Our third research question aimed to determine if the type of a bug impacts its fixing time. Not only we found that the type of a bug does significantly impact its fixing time, but we also found that, in average Types 2 and 4, stay open 111.26 days while Types 1 and 3 last for 77.36 days. Types 2 and 4 are 1.4 time longer to fix than Types 1 and 3.We therefore believe that, approaches aiming to predict the fixing time of a bug (e.g., [Pan07, BN11, ZGV13]) can highly benefit from accurately predicting the type of a bug and thereforebetter plan the required man-power to fix the bug. In summary, Types 2 and 4 account for 65% of the bugs and they are more complex, have a bigger impact and take longer to be fixed than Types 1 and 3 while being equivalent in terms of severity.

Our taxonomy aimed to analyse: (1) the proportion of each type of bugs; (2) the complexity of each type in terms of severity, reopening and duplication; (3) the required time to fix a bug depending on its type. The key findings are:

- Types 2 and 4 account for 65% of the bugs.
- Types 2 and 4 have a similar severity compared to Types 1 and 3.

- Types 2 and 4 are more complex (reopening) and have a bigger impact (duplicate) than Types 1 and 3.
- It takes more time to fix Types 2 and 4 than Types 1 and 3.

Our taxonomy and results can be built upon in order to classify past and new researches in several active areas such as bug reproduction and triaging, prediction of reopening, duplication, severity, files to fix and fixing time. Moreover, if one could predict the type of a bug at submission time, all these areas could be improved.
In order to ease all these future works we built `BUMPER` that we present in the next section.

3.2 BUMPER - Bug Meta-repository For Developers & Researchers

With the goal to support the research towards analyzing relationships between bugs and their fixes we constructed a dataset of 380 projects, more than 100,000 resolved/fixed and with 60,000 changesets that were involved in fixing them from Netbeans and The Apache Software foundation's software that is (1) searchable in natural language at research.mathieu-nayrolles.com/bumper, (2) contains clear relationships between the bug report and the code involved to fix it, (3) supports complex queries such as parent-child relationships, unions or disjunctions and (4) provide easy exports in json, csv and xml format.
In what follows, we will present the projects we selected. Then, we present the features related to the bugs and their fixes we integrate in BUMPER (BUg Metarepository for dEvelopers and Researchers) and how we construct our dataset. Then, we present the API, based on Apache Solr [Nay14], which allows the NLP search with practical examples before providing research opportunities based on our dataset.
However, to the best of our knowledge, no attempt has been made towards building a unified and online dataset where all the information related to a bug, or a fix can be easily accessed by researchers and engineers.

3.2.1 Data collection

Figure 12 illustrates our data collection and analysis process that we present here and discuss in more detail in the following subsections. First, we extract the raw data from the two bug report management systems used in this study (Bugzilla[1] and Jira[2]). The extracted data is consolidated in one database called BUMPER where we associate each bug report

[1]https://netbeans.org/bugzilla/
[2]https://issues.apache.org/jira/issues/?jql=

with its fix. The fixes are mined from different type of source versioning system. Indeed, Netbeans is based on mercurial[3] while we used the git[4] mirrors[5] for the Apache Foundation software.

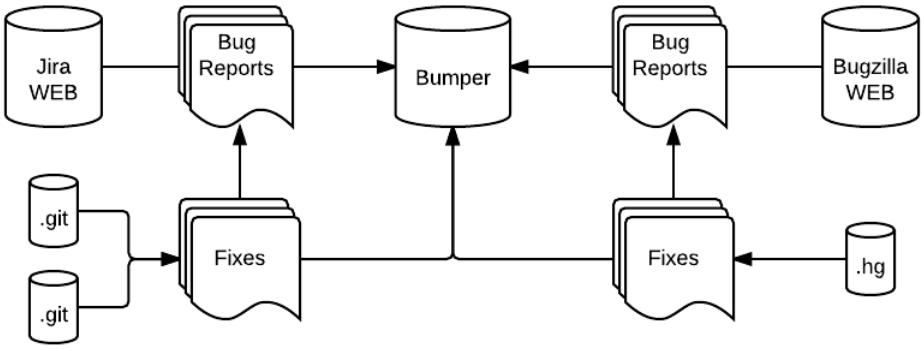

Figure 10: Overview of the bumper database construction.

We used the same datasets presented earlier in Table 2. We choose to use the same ones as these two datasets because they exposed a great diversity in programming languages, teams, localization, utility and maturity. Moreover, the used different tools, i.e. Bugzilla, JIRA, Git and Mercurial, and therefore, BUMPER is ready to host any other datasets that used any composition of these tools.

3.2.2 Architecture

`BUMPER` rely on a highly scalable architecture composed of two distinct servers as depicted in Figure 11. The first server, on the left, handles the web requests and runs three distinct components:

- Pound is a lightweight open source reverse proxy program and application firewall. It is also served us to decode to request to http. Translating an request to http and then, use this HTTP request instead of the one allow us to save the http's decryption time required at each step. Pound also acts as a load-balancing service for the lower levels.

- Translated requests are then handled to Varnish. Varnish is an HTTP accelerator designed for content-heavy and dynamic websites. What it does is caching request that come in and serve the answer from the cache is the cache is still valid.

[3] http://mercurial.selenic.com/
[4] http://git-scm.com/
[5] https://github.com/apache

- NginX (pronounced engine-x) is a web-server that has been developed with a particular focus on high concurrency, high performances and low memory usage.

On the second server, that concretely handles our data, we have the following items:

- Pound. Once again, we use pound here, for the exact same reasons.
- SolrCloud is the scalable version of Apache Solr where the data can be separated into shards (e.g chunk of manageable size). Each shard can be hosted on a different server, but it's still indexed in a central repository. Hence, we can guarantee a low query time while exponentially increasing the data.
- Lucene is the full text search engine powering Solr. Each Solr server has its own embedded engine.

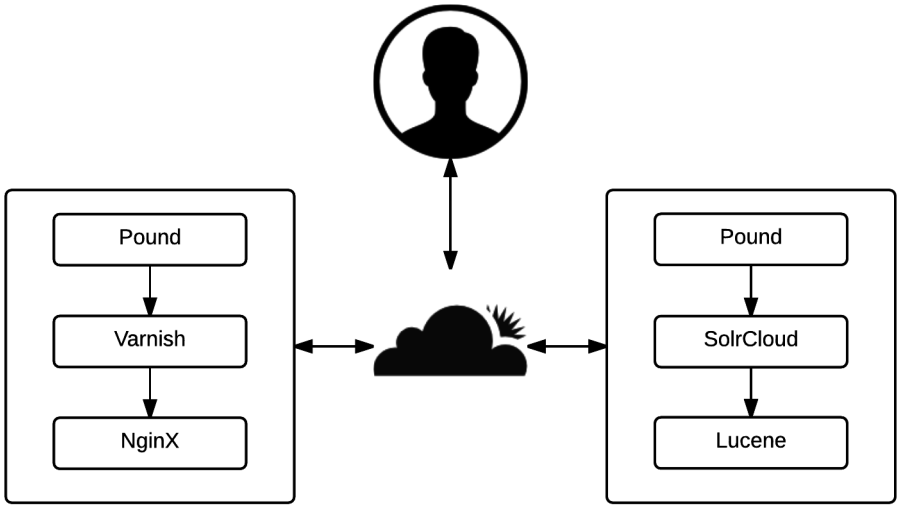

Figure 11: Overview of the bumper architecture.

Request from users to the servers and the communication between our servers are going through the CloudFlare network. CloudFlare acts as a content delivery network sitting between the users and the webserver. They also provide an extra level of caching and security.
To give the reader a glimpse about the performances that this unusual architecture can yield; we are able to request and display the result of a specific request in less than 100 ms while our two servers are, in fact, two virtual machines sharing an AMD Opteron (tm) Processor 6386 SE (1 core @ 2,000 MHz) and 1 GB of RAM.

3.2.3 UML Metamodel

Figure 12 presents the simplified `BUMPER` metamodel that we designed according to our bug taxonomy presented in section 4 and according to our future needs for `JCHARMING`, `RESSEMBLE` and `BIANCA`.

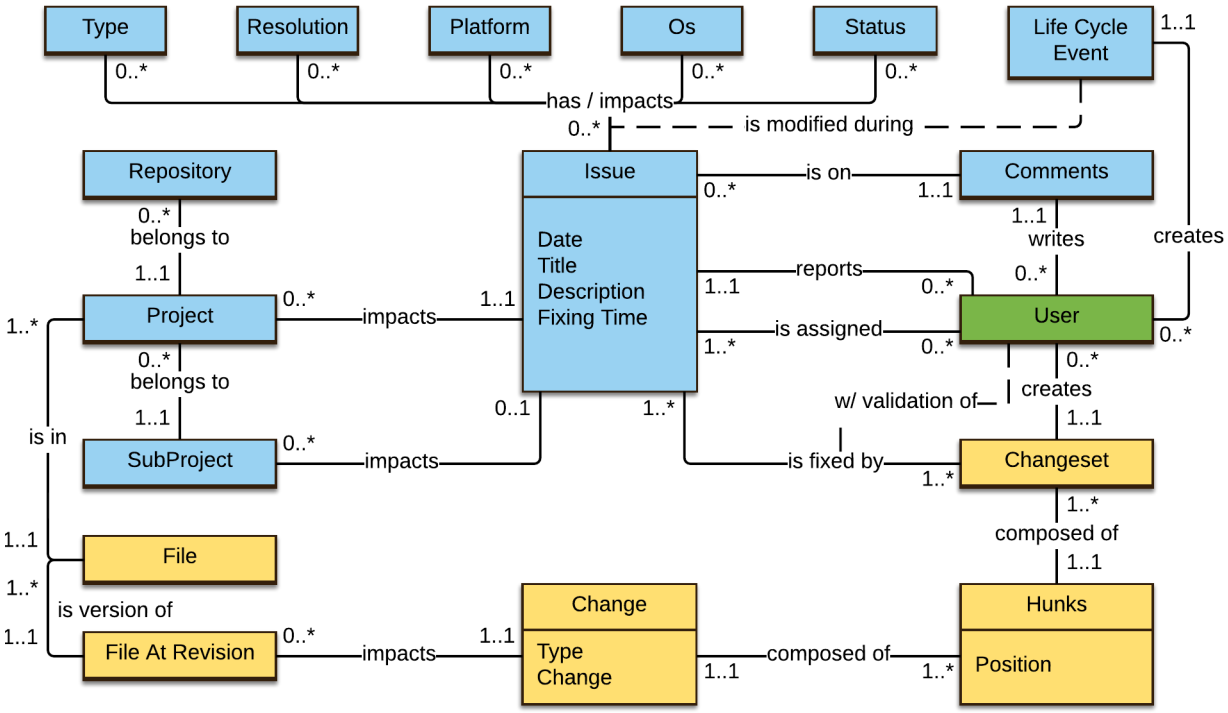

Figure 12: Overview of the bumper meta-model.

An *issue* (*task*) is characterized by a *date*, *title*, *description*, and a *fixing time*. They are reported (created) by and assigned to *users*. Also, *issues* (*tasks*) belong to *project* that are in *repository* and might be composed of *sub-projects*. *Users* can modify an *issue* (*task*) during *life cycle events* which impact the *type*, the *resolution*, the *platform*, the *OS* and the *status*. *Issues* (*tasks*) are resolved (implemented) by *changeset* 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*.

3.2.4 Features

In this section, we present the features of bug report and their fixes in details.

Bug Report

A bug report is characterized by the following features:

- ID: unique string id of the form bug_dataset_project_bug_id
- Dataset: the dataset of which the bug is extracted from.
- Type: The type help us to distinguish different type of entities in BUMPER, i.e the bugs, changesets and hunks. For bug report, the type is always set to BUG
- Date: The date at which the bug report has been submitted.
- Title: The title of the bug report.
- Project: The project that this bug affects.
- Sub_project: The sub-project that this bug affects.
- Full_name_project: The combination of the project and the sub-project.
- Version: the version of the project that this bug affects
- Impacted_platform: the platform that this bug affects
- Impacted_os: the operating system that this bug affects
- Bug_status: The status of the bug. As in bumper, our main concern is on the relationship between of fix and a bug, we only have RESOLVED bugs
- Resolution: How the bug was resolved. Once again, as we are interested in investigating the fixes and the bugs, we only have FIXED bugs.
- Reporter_pseudo: the pseudonym of the person who report the bug.
- Reporter_name: the name of the person who reported the bug
- Assigned_to_pseudo: the pseudonym of the person who have
- been assigned to fix this bug
- Assigned_to_name: the name of the person who have been assigned to fix this bug
- Bug_severity: the severity of a bug
- Description: the description of the bug the reporter gave

- Fixing_time: The time it took to fix the bug, i.e the elapsed time between the creation of the BR and its modification to resolve/fixed, in minutes
- Comment_nb: How many comments have been posted on the bug report system for that bug
- Comment: Contains one comment. A bug can have 0 or many comments
- File: A file qualified name that has been modified in order to fix a bug. A bug can have 0 (in case we did not find its related commit) or many files.

We selected this set of features for bug report as they are the ones that are analyzed in many past and recent studies. In addition, bugs can contain 0 or many .

Changesets

In this section, we present the features that characterize changeset entities in BUMPER.

- ID: the SHA1 hash
- User: the name and email of the person who submitted that commit
- Date: the date at which this commit has been fixed
- Summary: the commit message entered by the user
- File: The fully qualified name of a file modified on that commits. A changeset can have 1 or many files.
- Number_files: How many files have been modified in that commit
- Insertions: the number of inserted lines
- Deletions: the number of deleted lines
- Churns: the number of modified lines
- Hunks: the number of sets of consecutive changed lines
- Parent_bug: the id of the bug this changeset belongs to.

In addition, changesets contain one or many hunks.

Hunks

A hunks are a set of consecutive lines changed in a file in order. A set of hunks form a fix that can be scattered across one or many files. Knowing how many hunks a fixed required and what are the changes in each of them is useful, as explained by [2] to understand how many places developers have to go to fix a bug.
Hunks are composed of:

- ID: unique id based on the files, the insertion and the SHA1 of the commits
- Parent_changeset: the SHA1 of the Changeset this hunk belongs to
- Parent_bug: the id of the bug this hunk belongs to.
- Negative_churns: how many lines have been removed in that hunk
- Positive_churns: how many lines have been added in that hunk
- Insertion: the position in a file at which this hunk takes place.
- Change: One line that have been added or removed. A Hunk can contain one or many changes.

3.2.5 Application Program Interface (API)

BUMPER is available for engineers and researchers at **https://bumper-app.com** and take the form of a regular search engine. Bumper supports (1) natural language query, (2) parent-child relationships, query, (3) disjunctions and union between complex queries and (4) a straight forward export of query results in XML, CSV or JSON format.
Browsing BUMPER, the basic query mode, perform the following operation:

$$(type : BUG\ AND\ report\_t : (\text{“}YOUR\ TERMS\text{”}))\ OR\ (!parent\ which = type\text{“}BUG\text{”})\ fix\_t : \text{“}YOUR\ TERMS\text{”}) \tag{1}$$

The first part of the query component of the query retrieves all the bugs that contains the “$YOUR\ TERMS$” query in at least one its features by selecting type: BUG and report_t, which is an index composed of all the features of the bug, set to “$YOUR\ TERMS$”. Then, we merge this query with another one that reads
$(!parent\ which = type\text{“}BUG\text{”}) fix\_t :\ \text{“}YOUR\ TERMS\text{”})$. In this one, we retrieve the parent documents, i.e the bugs, of fixes that contains “$YOUR\ TERMS$” in their fix_t

index. The fix_t index is, as for the BUG, an index based on all the fields of changeset and hunk both. As a result, we search seamlessly in the bug report and their fixes in natural language.

As a more practical example, Figure 13 illustrate a query on https://bumper-app.com. The search term is "*Exception*" and we can see that 20,285 issues / tasks have been found in 25 ms This particular set of issues, displayed on the left side, match because they contain "*Exception*" in the issue report or in the source code modified to fix this issue (implement this task). Then on the right side of the screen, the selected issue (task) is displayed. We can see the basic characteristic of the issue (task) followed by comments and finally, the source code.

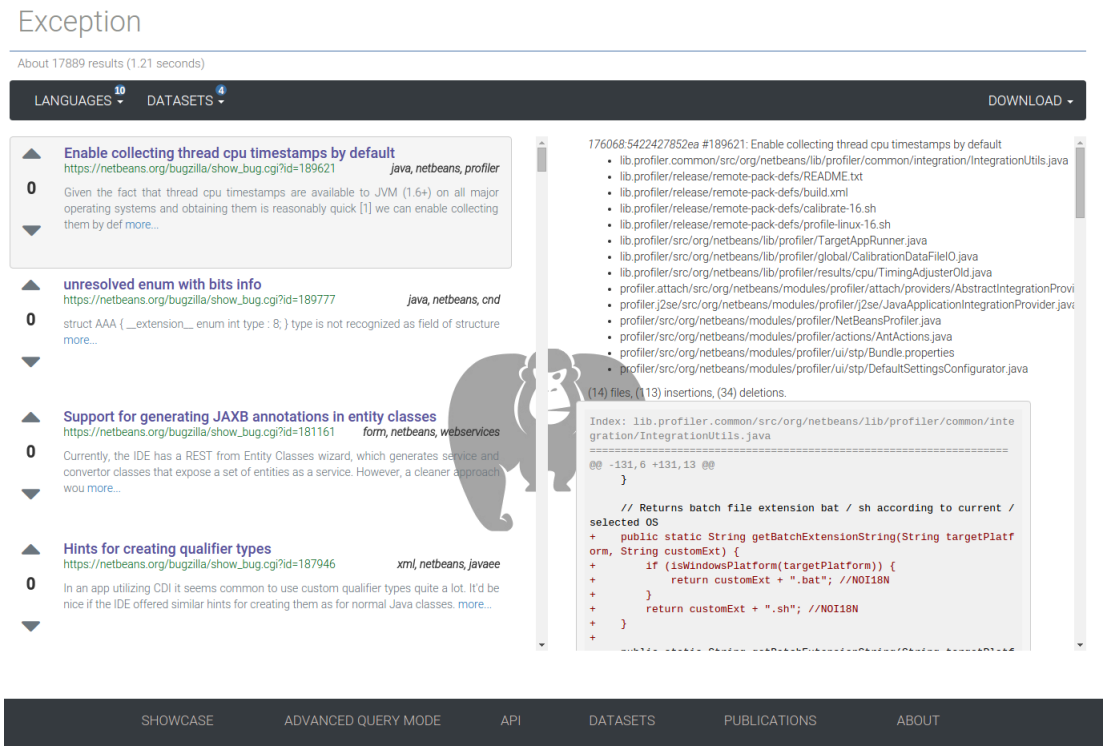

Figure 13: Screenshot of https://bumper-app.com with "Exception" as research.

Moreover, BUMPER supports AND, OR, NOR operators and provide results in order of seconds.

As we said before, BUMPER is based on Apache Solr which have an incredibly rich API that is available online[6].

`BUMPER` serves as data repositories for the upcoming approaches presented in the next sections.

[6] http://lucene.apache.org/solr/resources.html

3.3 JCHARMING - Java CrasH Automatic Reproduction by directed Model checkING

Field failures are challenging to reproduce because the data provided by the end users is often scarce. 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 [BJS+08]. 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.

In this section, we present an approach, called `JCHARMING` (Java CrasH Automatic Reproduction by directed Model checkING) that uses a combination of crash traces and model checking to automatically reproduce bugs that caused field failures. Unlike existing techniques, such as on-field record and in-house replay [NPC05, AKE08, JKXC10] or crash explanation [MSA04, CFS09] 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. While we do not filter any personal information that may appear in the crash trace, JCHARMING raises less privacy concerns than a tool recording every call or dump the content of the memory.

To assess the effiency of `JCHARMING` we try to reproduce issues contained in `BUMPER`.

3.3.1 Preliminaries

Model checking (also known as property checking) will, given a system (that could be software [VHB+03] or hardware based [Kro99]), check if the system meets a specification Spec by testing exhaustively all the states of the system under test (SUT), which can be represented by a Kripke [Kri63] structure:

$$SUT =< S, T, P > \tag{2}$$

where S is the set of states, $T \subseteq S * S$ represents the transitions between the states and P is the set of properties that each state satisfies. The SUT is said to satisfy a set of properties p when there exists a sequence of states transition x leading towards these properties. This can be written as:

$$(SUT, x) \models p \tag{3}$$

However, this only ensures that $\exists x$ such that p is reached at some point in the execution of the program and not that p holds nor that $\forall x$, p is satisfiable. In JCHARMING, SUTs are bound to a simple specification: they must not crash under a fair environment. In the framework of this study, we consider a fair environment as any environment where the transitions between the states represent the functionalities offered by the program. For example, in a fair environment, the program heap or other memory spaces cannot be modified. Without this fairness 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 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:

$$\forall x.(SUT, x) \models \neg c \tag{4}$$

If such a path exists (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 14, 15 and 16 to sketch the possible results of each approach. These figures depicts 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 $c_{i>2}$.

Figure 14 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 exhaustive. In our case, using testing would mean that the tester knows what to look for in order to detect the causes of the failure. We do not assume

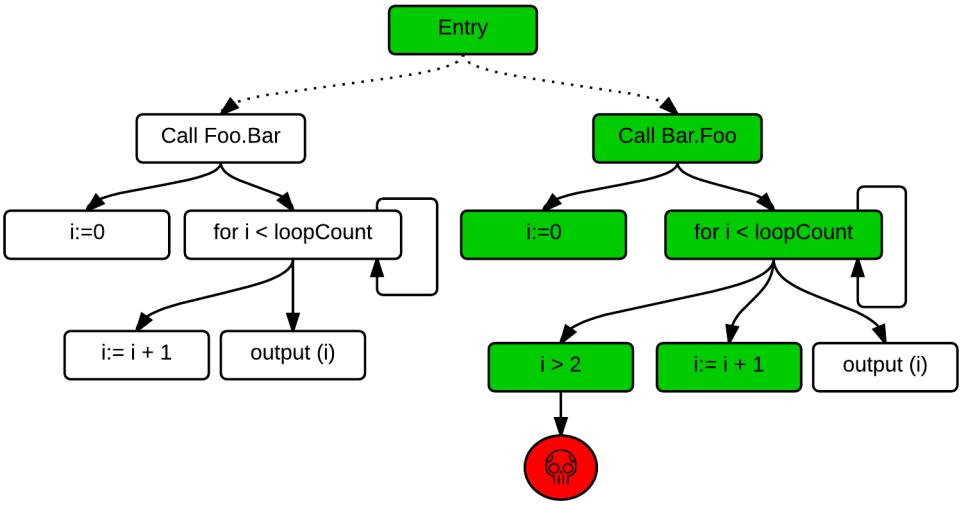

Figure 14: A toy program under testing

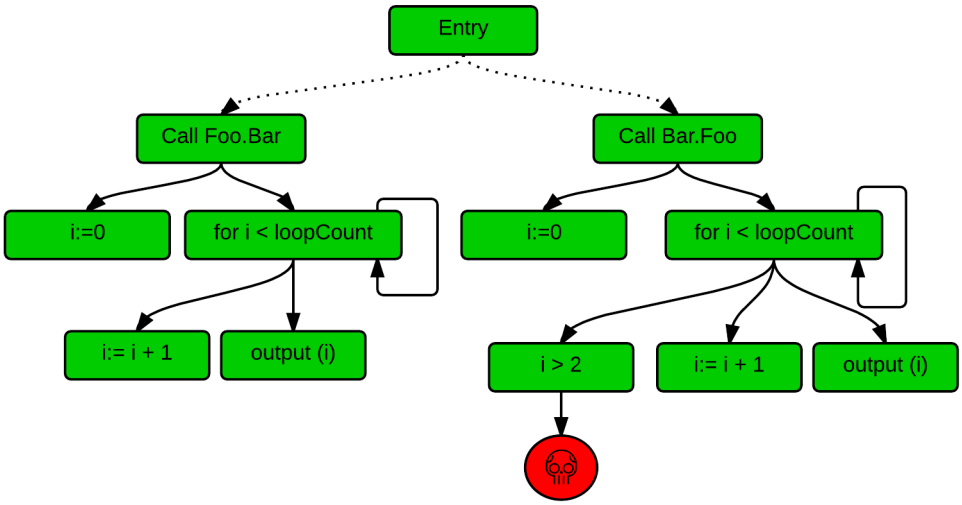

Figure 15: A toy program under model checking

this knowledge in JCHARMING.

Model checking, on the other hand, explores each and every state of the program (Figure 15), 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 [RM09]. Directed model checking use insightsgenerally heuristicsabout the SUT in order to reduce the number of states that need to be examined. Figure 16 explores only the states that may lead to a specific location, in our case, the location of the fault. The

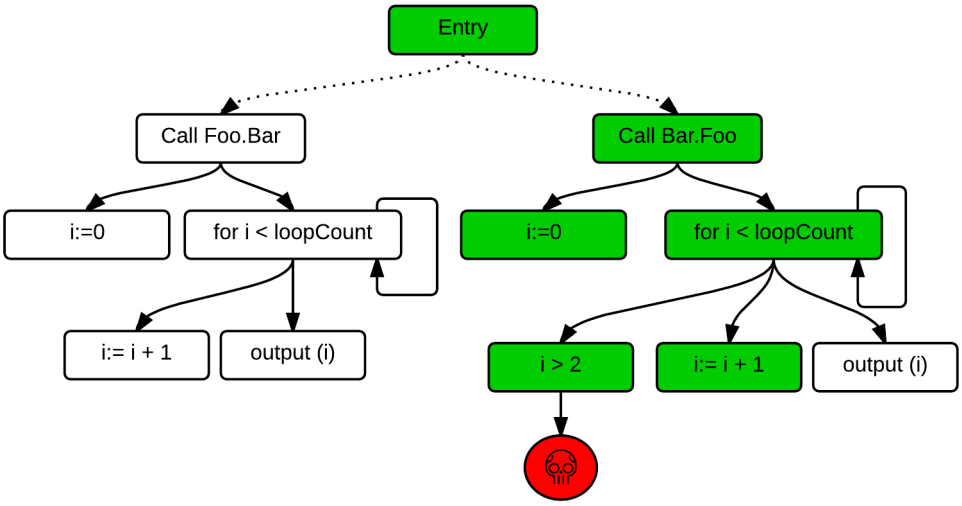

Figure 16: A toy program under directed model checking

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.

3.3.2 The JCHARMING Approach

Figure 17 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.

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

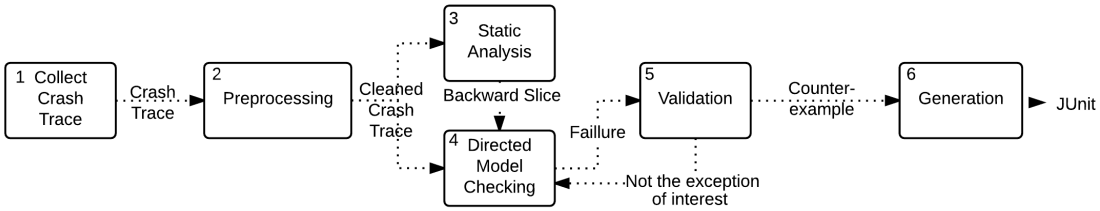

Figure 17: Overview of JCHARMING.

retrieved using a simple regular expression. Figure 18 shows an example of a crash trace that contains the exception thrown when executing the program depicted in Figures 14 to 16. The crash trace contains a call to the Bar.foo() methodthe crash location pointand calls to Java standard library functions (in this case, GUI methods because the program was launched using a GUI).

1.javax.activity.InvalidActivityException:loopTimes
should be ¡ 3
2. at Foo.bar(Foo.java:10)
3. at GUI.buttonActionPerformed(GUI.java:88)
4. at GUI.access0($GUI.java$: 85)
5.$atGUI$1.actionPerformed(GUI.java:57)
6. caused by java.lang.IndexOutOfBoundsException : 3
7. at scam.Foo.buggy(Foo.java:17)
8. and 4 more ...

Figure 18: Java InvalidActivityException is thrown in the Bar.Goo loop if the control variable is greater than 2.

As shown in Figure 18, 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 `scam.Foo.buggy` . This kind of crash traces reflects several nested try/catch blocks.

In addition, it is common in Java to have incomplete crash traces. According to the Java documentation [Ora11], line 8 of Figure 18 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.

Preprocessing

In the preprocessing step, we first reconstruct and reorganize the crash trace in order to address the problem of nested exceptions. Then, with the aim to obtain an optimal guidancefor our directed model checking engine, we remove frames that are out of our control. Frames out of our controls refer usually, but are not limited to, Java library methods and third party libraries. In Figure 18, 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 on- field JDK. 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.

Building the Backward Static Slice

For large systems, a crash trace does not necessary contain all the methods that have been executed starting from the entry point of the program (i.e., the main function) 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 18, this will be the function call `Bar.foo()`, which happens to be also the crash location point. To achieve this, we turn to static analysis by extracting a backward slice from the main function of the program to the `Bar.foo()` method.
A backward slice contains all possible branches that may lead to a point n from a point m as well as the definition of the variables that control these branches [De 01]. 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 $bslice_{[m \leftarrow n]}$.
We perform a static backward slice between each frame to compensate for possible missing information in the crash trace. More formally, the final static backward slice is represented as follows:

$$bslice_{[entry \leftarrow f_0]} = bslice_{[f_1 \leftarrow f_0]} \cup bslice_{[f_2 \leftarrow f_1]} \cup ... \cup bslice_{[f_n \leftarrow f_{n1}]} \cup bslice_{[entry \leftarrow f_n]} \quad (5)$$

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. More formally:

$$\bigcup_{i=0}^{entry} bslice_{[f_{i+1} \leftarrow f_i]} \subseteq bslice_{[entry \leftarrow f_0]} \tag{6}$$

Indeed, in Figure 19, the set of states allowing to reach f_0 from f_2 is greater than the set of states to reach f_1 from f_2 plus set of states to reach f_0 from f_1 . In this hypothetical example and assuming that z_2 is a prerequisite to f_2 then $bslice_{[entry \leftarrow f_0]} = \{f_0, f_1, f_2, z_0, z_1, z_2, z_3\}$ while $\cup_{i=0}^{n} bslice_{[f_{i+1} \leftarrow f_i]}$.

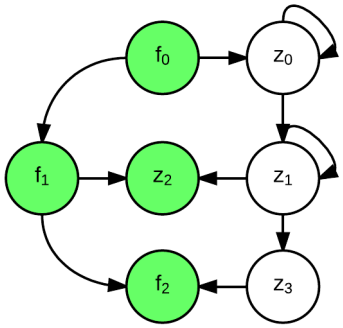

Figure 19: Hypothetical example representing $bslice_{[entry \leftarrow f_0]}$ Vs. $\cup_{i=0}^{n} bslice_{[f_{i+1} \leftarrow f_i]} = \{f_0, f_1, f_2, z_2\}$

In the worst case scenerio where there exists one and only one transition between each frame, which is very unlikely for real and complex systems, then $bslice_{[entry \leftarrow f_0]}$ 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 1 is a high level representation of how we compute the backward slice between each frame. The algorithm takes as input the pre-processed call trace, the byte code of the SUT, and the entry point. From line 1 to line 5, we initialize the different variables used by the algorithm. The main loop of the algorithm begins at line 6 and ends at line 15. In this loop, we compute the static slice between the current frame and the next one. If the computed static slice is not empty then we update the final backward slice with the newly

computed slice.

```
Data: Crash Stack, BCode, Entry Point
Result: BSolve
Frames frames ← extract frames from crash stack;
Int n ← size of frame;
Int offset ← 1;
Bslice bSlice ← ∅;
for i ← 0 to i < n && offset < n - 1 do
    BSlice currentBSlice ← backward slice from frames[i] to i + offset;
    if currentBSlice ≠ ∅ then
        bSlice ← bSlice ∪ currentBSlice;
        offset ← 1;
    else
        offset ← offset +1;
    end
end
```

Algorithm 1: High level algorithm computing the union of the slices

Using backward slicing, the search space of the model checker that processes the example of Figures 14 to 16 is given by the following expression:

$$\exists x. \begin{pmatrix} \bigcup_{i=0}^{entry} bslice_{[f_{i+1} \leftarrow f_i]} \subset SUT \\ x. \bigcup_{i=0}^{entry} bslice_{[f_{i+1} \leftarrow f_i]} \subset x.SUT \end{pmatrix} \models c_{i>2} \tag{7}$$

That is, there exists a sequence of states transitions x that satisfies $c_{i>2}$ where both the transitions and the states are entry elements of $\bigcup_{i=0}^{entry} bslice_{[f_{i+1} \leftarrow f_i]}$. Obviously, $c_{i>2}$ also needs to be included for the final static slice to be usable by the model checking engine. Consequently, the only frame that need to be untouched for the backward static slice to be meaningful is f_0.

Directed Model Checking

The model checking engine we use in this paper is called JPF (Java PathFinder) [VPK04], which is an extensible JVM for Java bytecode verification. This tool was first created as a front-end for the SPIN model checker [Hol97] in 1999 before being open- sourced in 2005. 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} . 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 and can be used to restore the entire program as it was the last time we choose 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 entry 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 $x.\bigcup_{i=0}^{entry} bslice_{[f_{i+1} \leftarrow f_i]} \subset x.SUT$.

Validation

To validate the result of directed model checking, we modify the *check properties* step that checks if the current sequence of states transitions x satisfies a set a property. If the current states transitions 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 [KZN13], 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. t is the percentage of identical frames between both crash traces.

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 exercise of 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. Although we can track back the internal state of objects at a specific time using JPF, it can be too computationally taxing to recreate only the objects needed to generate the bug. To overcome this, we use serialization techniques [OP99]. We take advantage of features offered by the XStream [Xst11] library which enables the

serialization and deserialization of any Java object even objects that do not implement the Java Serializable interface. We use the serialization when the model checker engine performs too many operations modifying the property of a given object. In such case, we serialize the last state of the object.

3.3.3 Case studies

In this section, we show the effectiveness of JCHARMING to reproduce bugs in seven open source systems[7] . The aim of the case study is 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?*

Targeted Systems

Table 9 shows the systems and their characteristics in terms of Kilo Line of Code (KLoC) and Number of Classes (NoC).

SUT	KLOC	NoC	Bug #ID
Ant	265	1233	38622, 41422
ArgoUML	58	1922	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	1412, 1359

Table 9: List of taget systems in terms of Kilo line of code (KLoC), number of classes (NoC) and Bug # ID

Apache Ant [Apa] is a popular command-line tool to build make files. While it is mainly known for Java applications, Apache Ant also allows building C and C++ applications. We choose to analyze Apache Ant because it has been used by other researchers in similar studies.

ArgoUML [Col] is one of the major players in 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.

[7]The bug reports used in this study and the result of the model checker are made available for download from research.mathieu- nayrolles.com/jcharming/

Dnsjava [Wel13] 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 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 [Obj05] 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 and it is a relatively large application.
Apache Log4j [The99] is a logging library for Java. This is not a very large library, but it is extensively used by thousands of programs. As other Apache projects, this tool is well maintained by a strong open source community and allows developers to submit bugs. The bugs which are in the bug report system of Log4j are, generally speaking, well documented and almost every bug contains a related crash trace and, therefore, it is a tool of interest to us.
MCT [NAS09] 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 [Apa14] 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.

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 a regular expression designed to find the pattern of a Java exception.

3.3.4 Results

Table 10 shows the results of JCHARMING in terms of Bug #ID, reproduction status, and execution time (in minutes) of directed model checking (DMC) and 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%. 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.

SUT	Bug #ID	Reprod.	Time DMC	Time MC
Ant	38622	Yes	25.4	-
	41422	No	-	-
ArgoUML	2558	Partial	10.6	-
	2603	Partial	9.4	-
	311	Yes	11.3	-
	1786	Partial	9.9	-
DnsJava	38	Yes	4	23
jFreeChart	434	Yes	27.3	-
	664	Partial	31.2	-
	916	Yes	26.4	-
Log4j	11570	Yes	12.1	-
	40212	Yes	15.8	-
	41186	Partial	16.7	-
	45335	No	-	-
	46271	Yes	13.9	-
	47912	Yes	12.3	-
	47957	No	-	-
MCT	440ed48	Yes	18.6	-
PDFBox	1412	Partial	19.7	-
	1359	No	-	-

Table 10: Effectiveness of JCHARMING using directed model checking (DMC) and model checking (MC) in minutes

As we can see in Table 10, we were able to reproduce 17 bugs out of 20 bugs either completely or partially (85ratio). The average time to reproduce a bug is 16 minutes. This result

demonstrates the effectiveness of our approach, more particularly, the use of backward slicing to create a manageable search space that guides adequately the model checking engine. We also believe that our approach is usable in practice since it is also time efficient. Among the 20 different bugs we have tested, we will describe one bug (chosen randomly) for each category (successfully reproduced, partially reproduced, and not reproduced) for further analysis.

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 following crash trace 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 (AWTEventMulticaster.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 .dispatchEventImp l(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)

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 neither the static slice nor 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.

Partially reproduced

As an example of a partially reproduced bug, we explore the 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 the 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.

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>

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) consists of few lines only.

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 reporter 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;236Al8E60000045C3A 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 [Apa00]. Struts is an open source MVC (Model View Controller) framework for building Java Web Application. JCHARMING examined the source code of Log4J for the crash location `struts.CPExceptionHandler.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 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.

While JCHARMING is effective at reproducing on-field failures in lab environment, we want to reduce their number in the coming year. To do so, we built `RESSEMBLE` and `BIANCA` that we present in the next two sections.

3.4 RESEMBLE - REcommendation System based on cochangE Mining at Block LEvel

`RESEMBLE` (REcommendation System based on cochangE Mining at Block LEvel) is a contextual recommendation system that will take place directly into developers' IDE. `RESEMBLE` will leverage the decades of data indexed by `BUMPER` to identify, on the fly, sub-optimum or hazardous code and display and actual solution to the problem.

3.4.1 Motivation

Most of the context-aware IDE [BGL+09, Joe] rely on web-search (e.g. querying google and specialized website, such as stack overflow) to fetch their information. Other approaches such as the *learn API by examples* ones [KCK11, MBFV13] do propose contextual examples, but programmers have to leave their IDE and search for a specific method or class in order to see examples on how to use it. While these approaches are useful and their authors were able to show their efficiency by using different groups of students, developers need to know what they don't know and search for it. Indeed, these approaches aggregate code samples and documentations but if one knows how to use a method, one will never use these services. Moreover, the said developer could be using this method in a suboptimal, dangerous or hazardous way.
By mining co-change at block level, normalizing code and comparing it to code sample known to be suboptimal, dangerous or hazardous because, for example, they led to the creation of an issue and a fix, `RESEMBLE` will be able to recommend modifications in order to improve the overall quality of a given piece of code.

3.4.2 The `RESEMBLE` approach

Figure 20 displays a general overview of the `RESEMBLE` approach. `RESEMBLE` mine changes via a plug-in that developers will have to install on their favorite IDE. We plan to support Netbeans, Eclipse and Sublime Text. Then, the `RESEMBLE` engine stands in two parts. The first part is local and runs on the developer's computer. This first part identifies the current block of code, normalize it, anonymize it and sent it to second part. The second part `RESEMBLE` will query `BUMPER` for related code sample and will build the contextual advices. In the following sections, we present the detailed steps of `RESEMBLE`.

Mining sequences of changes

In the data mining field, ARM is a well-established method for discovering co-occurrences between attributes in the objects of a large data set [G. 91, HHA97]. Plain associations have the form $X \rightarrow Y$, where X and Y, called the *antecedent* and the *consequent*, respectively, are sets of descriptors (purchases by a customer, network alarms, or any other general kind of events). Even though plain association rules could serve some relevant information, we are interested here in the sequences of changes that we believe will yield more precise result. Indeed, we think that similar modifications are often done in the same order by the same developer (e.g top to bottom or bottom to top). We, therefore, adopt a variant

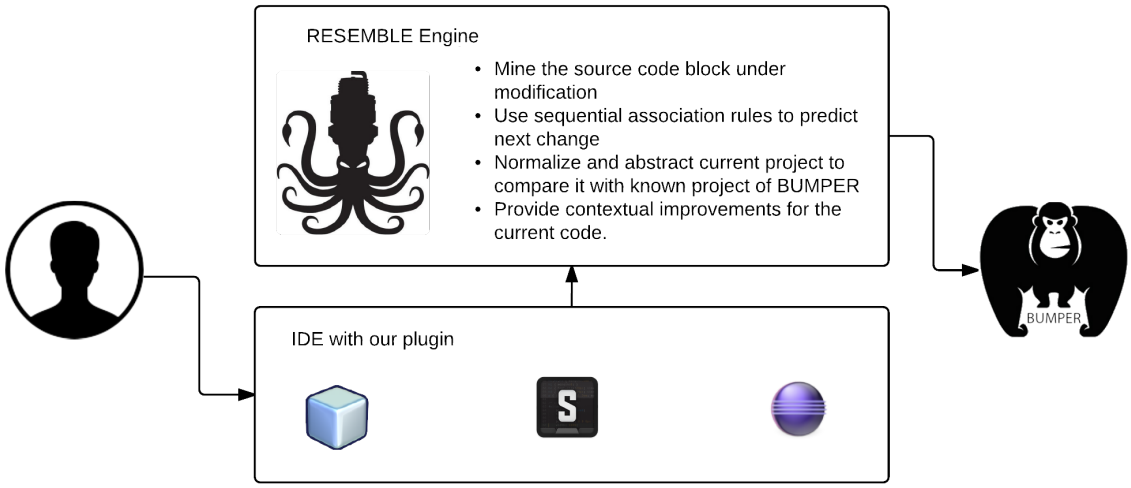

Figure 20: The RESEMBLE Approach

called sequential association rules in which both X and Y become sequences of descriptors. Moreover, our sequences follow a temporal order with the antecedent preceding the consequent. Rules of this type mined from changes reveal crucial information about the likelihood of blocks of code to be modified together in a programming session and, more importantly, in a specific order. For instance, a strong rule $Block_A$, $Block_B$ implies $Block_C$ would mean that after modifying $Block_A$ and then $Block_B$, there are good chances that the developer needs to modify $Block_C$. The conciseness of this example should not confuse the reader as in practical cases the sequences appearing in a rule can be of an arbitrary length. Furthermore, the strength of the rule is measured by the *confidence* metric. In probabilistic terms, it measures the conditional probability of C appearing down the line. Beside that, the significance of a rule, i.e. how many times it appears in the data, is provided by its *support* measure. To ensure only rules of potentially high interestingness are mined, the mining task is tuned by minimal thresholds to output only the sufficiently high scores for both metrics.

To extract the association rules from changes, two choices were possible. On one hand, sequential pattern mining and rule mining algorithms have been designed for structures that are slightly more general than the ones used here. In fact, sequential patterns are defined on transactions that represent sequences of sets. Efficient sequential pattern miners have been published, e.g. the PrefixSpan method [PHM+04]. On the other hand, sequence of changes do not compile to fully-blown sequential transactions as the underlying structures are mere sequences of individual elements. Such data has been known since at least the mid-90s but

received less attention by the data mining community, arguably because it is less challenging to mine. In the general data mining literature, mining from pure sequences, as opposed to sequences made of sets, has been addressed under the name of episode mining [HHA97]. Episodes are made of *events* and in a sense, code changes are events. Arguably the largest body of knowledge on the subject belongs to the web usage mining field: The input data is again a system log, yet this time the log of requests sent to a web server [PHMa00]. It is noteworthy that sequential patterns are more general than the pure sequence ones, hence mining algorithms designed for the former might prove to be less efficient when applied to the latter (as additional steps might be required for listing all significant set). Nevertheless, to jump-start our experimental study, we used a sequential pattern/rule miner that has the advantage to be freely available on the web[8]. Although it has not been optimized for pure sequences its performances are more than satisfactory.

Code Normalization

Normalizing code is the action of making its structure consistent throughout the program according to a defined model. In our case, we only normalize the block of code that are in the current sequential association rules. To do so, we improve and combine several technologies such as source code transformation [Cor06b, Cor06a], source code pretty-printing [RC08], flexible source code normalization [CR11].
More specifically, the code first goes to a pretty printer. A pretty-printer is a component that will slightly transform the code in order to obtain consistent control structure. Concretely, spaces will be added, accolade moved and tabulation added for an if, a *while* and others structures to always appear the same way, regardless of the programming language. Then, the code is normalized several times. Each normalization is different and targets specific feature of the code. For example, we have one normalization that removes completely the variable names and replace the types by the highest known object in the object oriented hierarchy before `Object` itself[9]. Another normalization only keeps the structures of the source code by normalizing both variables names and types.

Comparing normalization

Comparing different normalizations is the easiest step and can be done efficiently using using the longest common sequence (LCS) algorithm [Hir77]. If the LCS is above an user-defined threshold, then a two different behaviors can be observed:

[8] http://www.philippe-fournier-viger.com/spmf/
[9] For Java programs

- If the modified blocks' — and potentially the ones that are likely to be modified after according to our sequential association rules — normalizations match the normalizations of blocks of code that have been removed in past history. `RESEMBLE` recommends the replacing code as a better solution. In other words, if blocks A, B and C have been replaced by blocks A', B' and C' and blocks D, E and F normalizations match A, B and C, then, `RESEMBLE` recommends to transform D, E and F to look like A', B' and C'. This could lead to the introduction of software clones but we argue that (i) software clones are not always harmfull [**?**] and (ii) informing the developer that A', B' and C' exist could lead him to re-use these blocks.

- If the modified blocks' — and potentially of the ones that are likely to be modified after according to our sequential association rules — normalizations match the normalization of blocks of code that are present in the history. `RESEMBLE` computes the differences between the developer code and the history code in order to suggests what the developer have to do next.

3.4.3 Planned experiments

We did not started the experiments for `RESEMBLE` yet as the development of the IDE plugins is not complete. Nevertheless, we plan to conduct the following experiments:

- Full history test with Normalization 1.
- Full history test with Normalization 2.
- An human study where:
  - Developers use `RESEMBLE` in order to determine whether or not developers take into account our recommendations to avoid inserting defects in the code.
  - Developers use `RESEMBLE` in order to determine whether or not developers take into account our recommendations to complete their modification according to what we found in the history.
  - Rate the suggested solution in a scale from 1 to 10 in order to determine if the proposed change pattern does resolve the current problem.

We believe that `RESEMBLE` will be a real asset in a developer tool belt in order to ship better code in terms of quality, performances and security. However, as `RESEMBLE` aims to provide recommendations in real-time, it will not be able to be as exhaustive as an offline process. To fill this gap, we built `BIANCA` that we present in the next section.

3.5 BIANCA - Bug Insertion ANticipation by Clone Analysis at commit time

`BIANCA` (Bug Insertion ANticipation by Clone Analysis at commit time) is the final piece of the proposed ecosystem and, as such, the final failsafe that prevents developer to ship code that we know to be sub-optimum or to be at the very root of issues.

3.5.1 Motivation

Many tools exist to prevent a developer to ship *bad* code [Dan00, Hov07, MGDL10] or to identify *bad* code after executions (e.g in test or production environment) [NPMG12, NMV13]. However, these tools rely on metrics and rules to statically and/or dynamically identify sub-optimum code.

3.5.2 The `BIANCA` approach

`BIANCA` is different than the tool presented in the previous section because, as `RESEMBLE` it fetches its data into the common knowledge of millions of projects and developers. More specifically, `BIANCA` mines and analyzes the change patterns in commits and matches it against past commits known to have introduced a defect in the code (or that have just been replaced by better implementation). Figure 21 presents an overview of our approach.
`BIANCA` builds a model where each issue is represented by three versions of the same file. These three versions are stored in `BUMPER`. The first version n is called the *stable state* because the code of this version was used to fix an issue. The $n-1$ version, however, is called the *unstable state* as it was marked as containing an issue. Finally, the third version is called the *before state* and represents the file before the introduction of the bug. Hereafter, we refer to the *before state* as $n-2$. `BIANCA` extracts the change patterns form $n-2$ to $n-1$ and from $n-1$ to n. It aslo generates the changes to go from $n-2$ to n.
When a developer commits new modifications, `BIANCA` extracts the change pattern from the version n_{dev} (current version) and $n-1_{dev}$ (version before modification) of the developer's source code and compare this change patterns to known $n-2$ to $n-1$ patterns. If n_{dev} to $n-1_{dev}$ matches a $n-2$ to $n-1$ then it means that the developer is inserting a known defect in the source code. In such a case, `BIANCA` will propose the related $n-1$ to n pattern to the developer, so s/he could improve the source code and will show the related $n-2$ for the n pattern so the developer will learn how to s/he should have modified the code in the first place.
Moreover, if the issue was previously reproduced by `JCHARMING`, then `BIANCA` will display

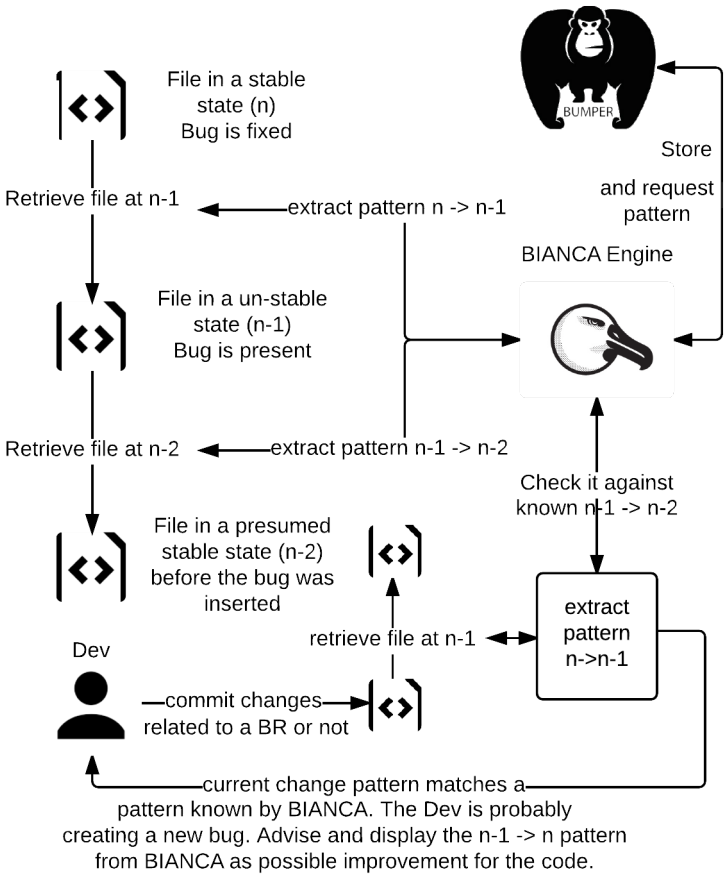

Figure 21: The BIANCA Approach

the step to reproduce it.

To extract the change patterns and compare them, we used the same technique as the one presented in sections 3.4.2 and 3.4.2 with new normalizations. The third and the fourth normalizations are removing all *less* important calls in the normalization one and two of `RESEMBLE`. We classify a call as less-important if, for example, it only does display-related functionalities such as generating HTML or printing something to the console. Finally, the fourth normalization will transform the code to an intermediate language of our own that will allow us to compare source code implemented in different programming languages.

Then, as in `RESEMBLE`, if the LCS is above a user-defined threshold, then a warning is raised by `BIANCA` alerting the developer that the commit is suspected to insert a defect. The given defect is shown to the developer and can either force the commit if s/he don't find the warning relevant or abort the commit.

We believe that the warning, alongside the previously mined change patterns and steps to reproduce the suspected default — provided by `JCHARMING`, if available — will statisfy

developers in terms of actionable inteligence. Thus, BIANCA could succeed, where other tools failed, at being used in industrial environment [LLS+13].

3.5.3 Early experiments

We have experimented the efficiency of BIANCA with the same datasets we used to build our bug taxonomy proposed in section 3.1.

Dataset	Fixed Issues	Commit	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 11: Datasets

We choosed to use the same datasets for several reasons. First of all, we spare the time needed to collect new datasets. Then, because these datasets contain a very large system mainly implemented in Java: Netbeans; and 349 independent Apache projects implemented in a very wide range of programming language. Consequently, these datasets allow us to test the efficiency of our different code normalizations.
We ran two different experiments using the two first normalizations we described in section 3.5.2. Both experiments consider only a few months of history, from April to August 2008. While this could hinder the pertinence of our results, these five months of history contain 167,597 commits related to bug fix. Consequently, we believe our results to be representative.
The first experiment yields the result presented by Figure 22.
With the first normalization, BIANCA raised 69,519 warnings out of 167,597 (41.5%) analyzed commits. Out of these 69,519, 13.4% turned out to be false positives. A false positive is a commit that have been tagged as introducing a bug by BIANCA but did not according to the history. However, false positives have to be dealt with carefully in this study as the commit might have introduced a bug but the bug could have not been reported yet.
In our second experiment, we used the second normalization and BIANCA raised 83,627 warnings out of 167,597 (48.89%) commits we analyze. However, the false positive rate increases to 21%. Figure 23 shows the results.

3.5.4 Planned experiments

BIANCA experiments are still in their early stage and we are still trying to improve our

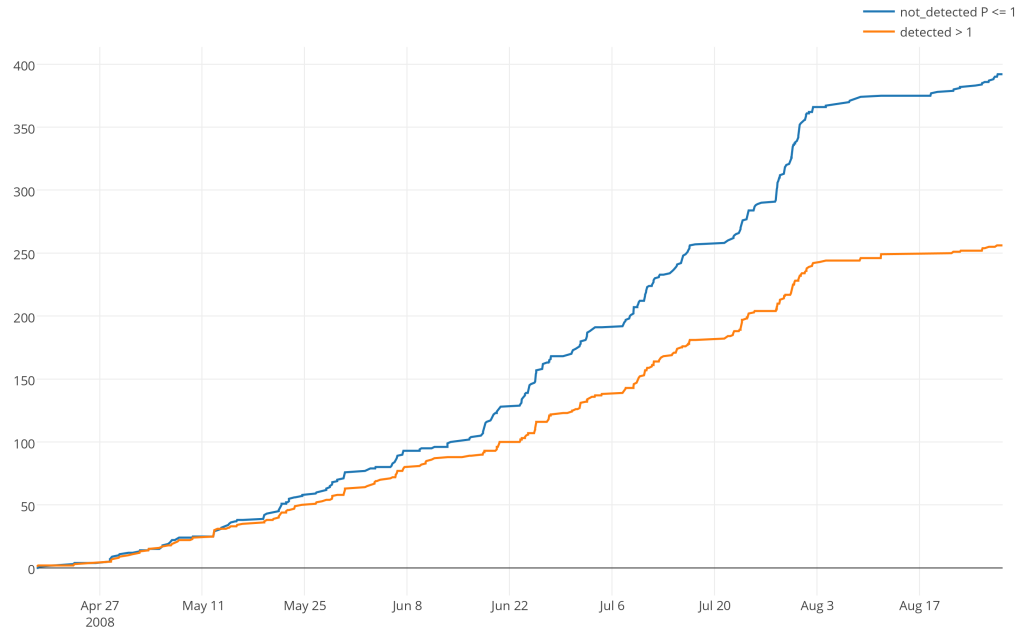

Figure 22: `BIANCA` warnings from April to August 2008 using the first normalization.

normalizations in order to reduce the false positive rate. In addition to these improvements, we want to conduct the following additional experiments:

- Full history test with Normalization 1.
- Full history test with Normalization 2.
- Full history test with Normalization 3.
- Full history test with Normalization 4.
- An human study where:
  - Developers use `BIANCA` in order to determine whether or not developers take into account our warnings or override them.
  - Rate the proposed solution in a scale from 1 to 10 in order to determine if the proposed change pattern does resolve the current problem.

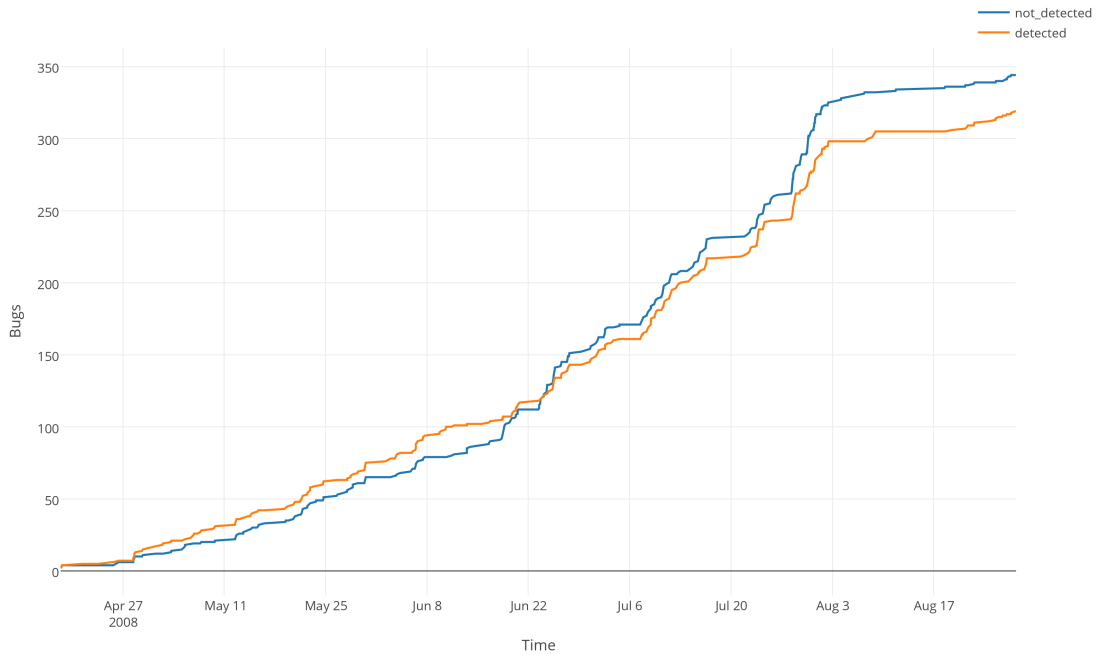

Figure 23: `BIANCA` warnings from April to August 2008 using the second normalization.

Chapter 4

Research Plan

In this chapter, we present the current state of research. Section 4.2 presents our current contributions and section 4.4 presents our publication plan.

4.1 Current State of Research

Chapter 3 presented the current state of our research. The completed parts are :

- To create a taxonomy of bug (section 3.1).
- To make software artifacts such as source code, issues, tasks, comments and more searchable using natural language and not structured query language (section 3.2).
- To efficiently reproduce field crashes in lab environment without raising any privacy concerns (section 3.3).
- To sequentially mine co-changes (section 3.4).
- To normalize source code (sections 3.4 and 3.5).
- To efficiently compare normalized source code (sections 3.4 and 3.5).
- To raise warning when a normalized source code matches another normalized source code known to have introduce a defect in an application (section 3.5).

4.2 Current Contribution

We already have one contribution referred in an international journal [NHLTL16]:

- Nayrolles, M. , Hamou-Lhadj, W., Tahar, S. Larsson, A. (2016). A Bug Reproduction Approach Based on Directed Model Checking and Crash Traces. Journal of Software: Evolution and Process. Wiley. (Accepted with revisions).

and three contributions referred in international conferences [NHLSL15, MHLN$^+$15, NHL16b, NHL16a]:

- Nayrolles, M. & Hamou-Lhadj, W. (2016). BUMPER: A Tool to Cope with Natural Language Search of Millions Bugs and Fixes. International Conference on Software Analysis, Evolution, and Reengineering - Tool Track (SANER'16). IEEE. (Accepted).
- Mathieu Nayrolles, Abdelwahab Hamou-Lhadj, Tahar Sofiene, and Alf Larsson. JCHARMING : A Bug Reproduction Approach Using Crash Traces and Directed Model Checking. In SANER'15, pages 101-110, 2015. (Best Paper Award).
- Maiga, A., Hamou-Lhadj, W., Nayrolles, M. , Sabor, K. & Larsson, A. (2015). An Empirical Study on the Handling of Crash Reports in a Large Software Company: An Experience Report. International Conference on Software Maintenance and Evolution (ICSME). (pp. 342-351). IEEE.

Finally, we also presented `BUMPER` during the 2015 Consortium for Software Engineering Research Fall Edition [NHL15]:

- Nayrolles, M. & Hamou-Lhadj, W. (2015). BUMPER: Bug Metarepository Search Engine for Developers and Researchers. Consortium for Software Engineering Research Fall.

4.3 Plan for short term work

As short term work, we want to complete the following parts:

- Recommend code in order to improve a sub-optimum code.
- Recommend code in order to complete an incomplete piece of code.
- Develop IDE plugins
- Develop new parsers (Github, Sourceforge) in order to improve:
  - The coverage of `BUMPER`.
  - The precision of `RESEMBLE` and `BIANCA`.
  - The accuracy of our Bug Taxonomy.

4.4 Publication Plan

This section presents our planned publications over the course of the next year. Depending on time constraints and possible invitations to journal extension, publication can be removed or added during the next years.

- **Publication 6**. `BIANCA` will be submitted to `International Symposium on Software Reliability Engineering, ISSRE 2016`.
- **Publication 7**. A derivate of `BIANCA` that detect clones at commit time will be submitted to `International Working Conference on Source Code Analysis and Manipulation, SCAM 2016`.
- **Publication 8**. `RESEMBLE` will be submitted to `International Conference Software Maintenance and Evolution, ICSME 2017`.
- **Publication 9**. Our proposed bug taxonomy will be submitted to `Transaction of Software Engineering, TSE 2017`.
- **Publication 10**. Our ecosystem as a whole, `BUMPER`, `JCHARMING`, `RESEMBLE` and `BIANCA` will be submitted to `Transaction of Software Engineering, TSE 2017`.
- **Thesis**. In parallel to publications 10 and 11, I plan to write my Ph.D thesis.

Chapter 5

Conclusion

The maintenance and evolution of complex software systems account for more than 70% software's life cycle. Hundreds of papers have been published with the aim to improve our knowledge of these processes in terms of issue triaging, issue prediction, duplicate issue detection, issue reproduction and co-changes prediction. All these publications gave meaning to the millions of issues that can be found in open source issue & project and revision management systems. Context-aware IDE and think tank in open source architecture ([AW12]) open the path to approaches that support developers during their programming sessions by leveraging past indexed knowledge and past architectures.

In this research proposal, we first presented the most influential papers in the different fields our work lies on in Chapter 2. Chapter 3 presented our proposal in details while chapter 4 detailed our attempt planning.

More specifically, in Chapter 3, we presented four approaches: `BUMPER`, `JCHARMING`, `RESEMBLE`, `BIANCA`. Also, we proposed a taxonomy of bugs. When combined into `pErICOPE` (Ecosystem Improve source COde during Programming session with real-time mining of common knowlEdge), these tools (i) provide the possibility to search related software artifacts using natural language, (ii) accurately reproduce field-crash in lab environment, (iii) recommend improvement or completion of current block of code and (iv) prevent the introduction of clones / issues at commit time.

`BUMPER` has been designed to handle heavy traffic while `JCHARMING` can reproduce 85% of real-world issues we submitted to it. On its side `BIANCA` is able to flag 41.5% and 48.89% of commit introducing a bug as dangerous with 13.4% and 21% of false positive using two code different code normalization, respectively.

Our future works, according to our publication plan described in section 4.4, are as follows. First, we want to improve the performances of `BIANCA` in terms of false positives. Then,

create the IDE plugin that will support `RESEMBLE`. Finally, we want to refine our taxonomy by including as many as datasets as possible.

Bibliography

[ACC+02] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, 28(10):970–983, oct 2002.

[AHM06] John Anvik, Lyndon Hiew, and Gail C Murphy. Who should fix this bug? In *Proceeding of the 28th international conference on Software engineering - ICSE '06*, page 361, New York, New York, USA, may 2006. ACM Press.

[AKE08] Shay Artzi, Sunghun Kim, and Michael D Ernst. Recrash: Making software failures reproducible by preserving object states. In *Proceedings of the 22nd European Conference on Object-Oriented Programming*, pages 542–565, 2008.

[Apa] Apache Software Foundation. Apache Ant.

[Apa00] Apache Software Foundation. Apache Struts Project, 2000.

[Apa14] Apache Software Foundation. Apache PDFBox — A Java PDF Library, 2014.

[Arm13] Tavish Armstrong. *The Performance of Open Source Applications*. 2013.

[AW12] Brown Amy and Greg Wilson. *The Architecture of Open Source Applications*. CreativeCommons, 2012.

[BBM96] V.R. Basili, L.C. Briand, and W.L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761, 1996.

[BBR+10] Adrian Bachmann, Christian Bird, Foyzur Rahman, Premkumar Devanbu, and Abraham Bernstein. The missing links. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering - FSE '10*, page 97, New York, New York, USA, nov 2010. ACM Press.

[BDW99] L.C. Briand, J.W. Daly, and J.K. Wust. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering*, 25(1):91–121, 1999.

[BG12] Amy Brown and Wilson Greg. *The Architecture of Open Source Applications, Volume II.* 2012.

[BGL+09] Joel Brandt, Philip J. Guo, Joel Lewenstein, Mira Dontcheva, and Scott R. Klemmer. Two studies of opportunistic programming. In *Proceedings of the 27th international conference on Human factors in computing systems - CHI 09*, page 1589, New York, New York, USA, apr 2009. ACM Press.

[BJS+08] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. What makes a good bug report? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, page 308, New York, New York, USA, 2008. ACM Press.

[BN11] Pamela Bhattacharya and Iulian Neamtiu. Bug-fix time prediction models. In *Proceeding of the 8th working conference on Mining software repositories - MSR '11*, page 207, New York, New York, USA, may 2011. ACM Press.

[BPZ08] Nicolas Bettenburg, Rahul Premraj, and Thomas Zimmermann. Duplicate bug reports considered harmful ... really? In *2008 IEEE International Conference on Software Maintenance*, pages 337–345. IEEE, 2008.

[Bur03] Oliver Burn. Checkstyle, 2003.

[BvdH13] Gerald Bortis and Andre van der Hoek. PorchLight: A tag-based approach to bug triaging. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 342–351. IEEE, may 2013.

[CFS09] Satish Chandra, Stephen J Fink, and Manu Sridharan. Snugglebug: a powerful approach to weakest preconditions. In *ACM Sigplan Notices*, volume 44, pages 363–374. ACM, 2009.

[Che13] Ning Chen. *Star: stack trace based automatic crash reproduction.* PhD thesis, The Hong Kong University of Science and Technology, 2013.

[CK94] S.R. Chidamber and C.F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, jun 1994.

[CNSH14] Tse-hsun Chen, Meiyappan Nagappan, Emad Shihab, and Ahmed E Hassan. An Empirical Study of Dormant Bugs Categories and Subject Descriptors. In *Mining Software Repository*, pages 82–91, 2014.

[CO07] James Clause and Alessandro Orso. A Technique for Enabling and Supporting Debugging of Field Failures. In *Proceedings of the 29th International Conference on Software Engineering*, pages 261–270, 2007.

[Col] CollabNet. Tigris.org: Open Source Software Engineering.

[Cor] Michael W. Godfrey Cory Kapser. Toward a Taxonomy of Clones in Source Code: A Case Study.

[Cor06a] James R. Cordy. Source transformation, analysis and generation in TXL. In *Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation - PEPM '06*, page 1, New York, New York, USA, jan 2006. ACM Press.

[Cor06b] James R. Cordy. The TXL source transformation language. *Science of Computer Programming*, 61(3):190–210, aug 2006.

[CR11] James R. Cordy and Chanchal K. Roy. The NiCad Clone Detector. In *2011 IEEE 19th International Conference on Program Comprehension*, pages 219–220. IEEE, jun 2011.

[Dan00] Andreas Dangel. PMD, 2000.

[De 01] Andrea De Lucia. Program slicing: Methods and applications. In *International Working Conference on Source Code Analysis and Manipulation*, page 144. IEEE Computer Society, 2001.

[DM06] Bruno Dutertre and Leonardo De Moura. The yices smt solver. *Tool paper at http://yices.csl.sri.com/tool-paper.pdf*, 2(2), 2006.

[DMT13] Anthony Demange, Naouel Moha, and Guy Tremblay. Detection of SOA Patterns. In *International Conference on Service-Oriented Computing*, pages 114–130, 2013.

[EMM01] Khaled El Emam, Walcelio Melo, and Javam C. Machado. The prediction of faulty classes using object-oriented design metrics. *Journal of Systems and Software*, 56(1):63–75, feb 2001.

[G. 91] G. Piatetsky-Shapiro. Discovery, analysis and presentation of strong rules. *Knowledge Discovery in Databases*, pages 229–249, jan 1991.

[GFS05] T. Gyimothy, R. Ferenc, and I. Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering*, 31(10):897–910, oct 2005.

[GHJV08] Eric Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns*. Addison Wesley, 2008.

[Has09] Ahmed E. Hassan. Predicting faults using the complexity of code changes. In *2009 IEEE 31st International Conference on Software Engineering*, pages 78–88. IEEE, may 2009.

[HGGBR08] Israel Herraiz, Daniel M. German, Jesus M. Gonzalez-Barahona, and Gregorio Robles. Towards a simplification of the bug report form in eclipse. In *Proceedings of the 2008 international workshop on Mining software repositories - MSR '08*, page 145, New York, New York, USA, may 2008. ACM Press.

[HH05] A.E. Hassan and R.C. Holt. The top ten list: dynamic fault prediction. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 263–272. IEEE, 2005.

[HHA97] MANNILA HEIKKI, TOIVONEN HANNU, and VERKAMO A. INKERI. Discovery of Frequent Episodes in Event Sequences. *Data Mining and Knowledge Discovery*, 289:259–289, 1997.

[Hir77] Daniel S Hirschberg. Algorithms for the longest common subsequence problem. *Journal of the ACM (JACM)*, 24(4):664–675, 1977.

[HNH15] H Hemmati, M Nagappan, and Ae Hassan. Investigating the effect of defect co-fix on quality assurance resource allocation: A search-based approach. *Journal of Systems and Software*, 00:1–18, 2015.

[Hol97] Gerard J Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[Hov07] David Hovemeyer. FindBugs, 2007.

[HR02] Health, Social and Economics Research. The Economic Impacts of Inadequate Infrastructure for Software Testing. Technical report, 2002.

[JKXC10] Hojun Jaygarl, Sunghun Kim, Tao Xie, and Carl K Chang. OCAT: Object Capture based Automated Testing. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, pages 159–170, 2010.

[JKZ09] Gaeul Jeong, Sunghun Kim, and Thomas Zimmermann. 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*, page 111, New York, New York, USA, aug 2009. ACM Press.

[JLL+12] Shujuan Jiang, Wei Li, Haiyang Li, Yanmei Zhang, Hongchang Zhang, and Yingqi Liu. Fault Localization for Null Pointer Exception Based on Stack Trace and Program Slicing. *2012 12th International Conference on Quality Software*, pages 9–12, aug 2012.

[JO12] Wei Jin and Alessandro Orso. BugRedux: Reproducing field failures for in-house debugging. In *Proceedings of the 34th International Conference on Software Engineering, IEEE*, pages 474–484. Ieee, jun 2012.

[JO13] Wei Jin and Alessandro Orso. F3: fault localization for field failures. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 213–223, New York, New York, USA, 2013. ACM Press.

[Joe] Joel Lewenstein Scott R Klemmer Mira Dontcheva Joel Brandt Philip J. Guo. Opportunistic Programming: Writing Code to Prototype, Ideate, and Discover.

[JW08] Nicholas Jalbert and Westley Weimer. Automated duplicate detection for bug tracking systems. In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, pages 52–61. IEEE, 2008.

[KCK11] Miryung Kim, Dongxiang Cai, and Sunghun Kim. An empirical investigation into the role of API-level refactorings during software evolution. *Proceeding of the 33rd international conference on Software engineering - ICSE '11*, page 151, 2011.

[KCZH11] Foutse Khomh, Brian Chan, Ying Zou, and Ahmed E. Hassan. An Entropy Evaluation Approach for Triaging Field Crashes: A Case Study of Mozilla Firefox. In *2011 18th Working Conference on Reverse Engineering*, pages 261–270. IEEE, oct 2011.

[Kri63] Saul A. Kripke. Semantical Considerations on Modal Logic. *Acta Philosophica Fennica*, 16(1963):83–94, 1963.

[Kro99] Thomas Kropf. *Introduction to formal hardware verification*. Springer, 1999.

[KTM+13] Dongsun Kim, Yida Tao, Student Member, Sunghun Kim, and Andreas Zeller. Where Should We Fix This Bug? A Two-Phase Recommendation Model. *Transaction on Software Engineering*, 39(11):1597–1610, 2013.

[KWM+11] Dongsun Kim, Xinming Wang, Student Member, Sunghun Kim, Andreas Zeller, S C Cheung, Senior Member, and Sooyong Park. Which Crashes Should I Fix First?: Predicting Top Crashes at an Early Stage to Prioritize Debugging Efforts. *TRANSACTIONS ON SOFTWARE ENGINEERING*, 37(3):430–447, 2011.

[KZN13] Sunghun Kim, Thomas Zimmermann, and Nachiappan Nagappan. Crash Graphs: An Aggregated View of Multiple Crashes to Improve Crash Triage. In *International Conference on Dependable Systems and Networks (DSN)*, pages 486–493, 2013.

[KZPJ06] Sunghun Kim, Thomas Zimmermann, Kai Pan, and E. Jr. Whitehead. Automatic Identification of Bug-Introducing Changes. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, pages 81–90. IEEE, 2006.

[KZWG11] Sunghun Kim, Hongyu Zhang, Rongxin Wu, and Liang Gong. Dealing with noise in defect prediction. *Proceeding of the 33rd international conference on Software engineering - ICSE '11*, page 481, 2011.

[KZWZ07a] Sunghun Kim, Thomas Zimmermann, E. James Whitehead Jr., and Andreas Zeller. Predicting Faults from Cached History. In *29th International Conference on Software Engineering*, pages 489–498. IEEE, may 2007.

[KZWZ07b] Sunghun Kim, Thomas Zimmermann, E. James Whitehead Jr., and Andreas Zeller. Predicting Faults from Cached History. In *29th International Conference on Software Engineering (ICSE'07)*, pages 489–498. IEEE, may 2007.

[LDGG10] Ahmed Lamkanfi, Serge Demeyer, Emanuel Giger, and Bart Goethals. Predicting the severity of a reported bug. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 1–10. IEEE, may 2010.

[LDSV11] Ahmed Lamkanfi, Serge Demeyer, Quinten David Soetens, and Tim Verdonck. Comparing Mining Algorithms for Predicting the Severity of a Reported Bug. In *2011 15th European Conference on Software Maintenance and Reengineering*, pages 249–258. IEEE, mar 2011.

[Leh80] Meir M Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980.

[LLS+13] Chris Lewis, Zhongpeng Lin, Caitlin Sadowski, Xiaoyan Zhu, Rong Ou, and E. James Whitehead Jr. Does bug prediction support human developers? findings from a google case study. pages 372–381, may 2013.

[LNH+11] Taek Lee, Jaechang Nam, DongGyun Han, Sunghun Kim, and Hoh Peter In. Micro interaction metrics for defect prediction. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering - SIGSOFT/FSE '11*, page 311, New York, New York, USA, 2011. ACM Press.

[Lo13] D. Lo. A Comparative Study of Supervised Learning Algorithms for Re-opened Bug Prediction. In *2013 17th European Conference on Software Maintenance and Reengineering*, pages 331–334. IEEE, mar 2013.

[MANH14] Shane Mcintosh, Bram Adams, Meiyappan Nagappan, and Ahmed E Hassan. Mining Co-Change Information to Understand when Build Changes are Necessary. In *Software Maintenance and Evolution (ICSME)*, 2014.

[MBFV13] Joao Eduardo Montandon, Hudson Borges, Daniel Felix, and Marco Tulio Valente. Documenting APIs with examples: Lessons learned with the APIMiner platform. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 401–408. IEEE, oct 2013.

[MGDL10] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. Le Meur. DECOR: A Method for the Specification and Detection of Code and Design Smells. *IEEE Transactions on Software Engineering*, 36(1):20–36, jan 2010.

[MHLN+15] A. Maiga, W. Hamou-Lhadj, M. Nayrolles, K. Sabor, and A. Larsson. An Empirical Study on the Handling of Crash Reports in a Large Software Company: An Experience Report. In *International Conference on Software Maintenance and Evolution (ICSME)*, pages 342–351. IEEE, 2015.

[MPN+12] Naouel; Moha, Francis; Palma, Mathieu; Nayrolles, Benjamin; Joyen-Conseil, Yann-Gaël; Guéhéneuc, Benoit; Baudry, and Jean-Marc; Jézéquel. Specification and Detection of SOA Antipatterns. *International Conference on Service Oriented Computing*, pages 1–16, 2012.

[MSA04] Roman Manevich, Manu Sridharan, and Stephen Adams. PSE: explaining program failures via postmortem static analysis. In *ACM SIGSOFT Software Engineering Notes*, volume 29, page 63. ACM, nov 2004.

[NAS09] NASA. Open Mission Control Technologies, 2009.

[Nay13a] Mathieu; Nayrolles. *Improving SOA Antipattern Detection in Service Based Systems by Mining Execution Traces*. PhD thesis, 2013.

[Nay13b] Mathieu; Nayrolles. *Instant Magento Performance Optimization How-to*. 2013.

[Nay14] Mathieu Nayrolles. *Mastering Apache Solr - A Practical Guide to Get to Grips with Apache Solr*. 2014.

[NB05a] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *Proceedings. 27th International Conference on Software Engineering, 2005.*, pages 284–292. IEEe, 2005.

[NB05b] Nachiappan Nagappan and Thomas Ball. Static analysis tools as early indicators of pre-release defect density. In *Proceedings of the 27th international conference on Software engineering - ICSE '05*, page 580, New York, New York, USA, may 2005. ACM Press.

[NBZ06] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. Mining metrics to predict component failures. In *Proceeding of the 28th international conference on Software engineering - ICSE '06*, page 452, New York, New York, USA, may 2006. ACM Press.

[NHL15] M. Nayrolles and W. Hamou-Lhadj. BUMPER: Bug Metarepository Search Engine for Developers and Researchers. Consortium for Software Engineering Research Fall., 2015.

[NHL16a] M. Nayrolles and W Hamou-Lhadj. A Bug-Fix Metarepository for Developers and Researcher. In *Mining Software Repositories - Data Track (MSR'16)*. IEEE, 2016.

[NHL16b] M. Nayrolles and W. Hamou-Lhadj. BUMPER: A Tool to Cope with Natural Language Search of Millions Bugs and Fixes. In *International Conference on Software Analysis, Evolution, and Reengineering - Tool Track (SANER'16)*. IEEE, 2016.

[NHLSL15] Mathieu Nayrolles, Abdelwahab Hamou-Lhadj, Tahar Sofiene, and Alf Larsson. 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, IEEE, 2015*, pages 101–110, 2015.

[NHLTL16] M. Nayrolles, W. Hamou-Lhadj, S. Tahar, and A. Larsson. A Bug Reproduction Approach Based on Directed Model Checking and Crash Traces. *Journal of Software: Evolution and Process. Wiley (Accepted with revisions).*, 2016.

[NMV13] Mathieu Nayrolles, Naouel Moha, and Petko Valtchev. Improving SOA Antipatterns Detection in Service Based Systems by Mining Execution Traces. In *Working Conference on Reverse Engineering*, number i, pages 321–330. IEEE, 2013.

[NNN+12] Anh Tuan Nguyen, Tung Thanh Nguyen, Tien N. Nguyen, David Lo, and Chengnian Sun. 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*, page 70, New York, New York, USA, 2012. ACM Press.

[NPC05] Satish Narayanasamy, Gilles Pokam, and Brad Calder. BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging. In *Proceedings of the 32nd annual International Symposium on Computer Architecture*, volume 33, pages 284–295. ACM, may 2005.

[NPMG12] Mathieu; Nayrolles, Francis; Palma, Naouel; Moha, and Yann-Gaël Guéhéneuc. SODA : A Tool Support for the Detection of SOA Antipatterns. In *International Conference on Service Oriented Computing LNCS 7759*, pages 451–456. Springer, 2012.

[Obj05] Object Refinery Limited. JFreeChart, 2005.

[OP99] L. Opyrchal and A. Prakash. Efficient Object Serialization in Java Lukasz Opyrchal and Atul Prakash. In *Proceedings. 19th IEEE International Conference on Distributed Computing Systems*, pages 96–101, 1999.

[Ora11] Oracle. Throwable (Java Plateform SE6), 2011.

[OWB05] T.J. Ostrand, E.J. Weyuker, and R.M. Bell. Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engineering*, 31(4):340–355, apr 2005.

[Pal13] Francis Palma. *Detection of SOA Antipatterns.* PhD thesis, Ecole Polytechnique de Montreal, 2013.

[Pan07] Lucas D. Panjer. Predicting Eclipse Bug Lifetimes. In *Fourth International Workshop on Mining Software Repositories (MSR'07:ICSE Workshops 2007)*, pages 29–29. IEEE, may 2007.

[PHM+04] Jian Pei, Jiawei Han, Senior Member, Behzad Mortazavi-asl, Jianyong Wang, Helen Pinto, and Qiming Chen. Mining Sequential Patterns by Pattern-Growth : The PrefixSpan Approach. *IEEE Transactions on Knowledge and Data Engineering*, 16(10):1424—-1440, 2004.

[PHMa00] Jian Pei, Jiawei Han, and Behzad Mortazavi-asl. Mining Access Patterns Efficiently from Web Logs *. In *Knowledge Discovery and Data Mining. Current Issues and New Applications*, volume 0, pages 396–407, 2000.

[PO11] Chris Parnin and Alessandro Orso. Are automated debugging techniques actually helping programmers? *Proceedings of the 2011 International Symposium on Software Testing and Analysis - ISSTA '11*, page 199, 2011.

[Pre05] Roger S. Pressman. *Software Engineering: A Practitioner's Approach.* Palgrave Macmillan, 2005.

[PS93] Perry, Dewayne E. and Carol S. Stieg. Software faults in evolving a large, real-time system: a case study. In *Software EngineeringESEC*, pages 48–67, 1993.

[RAN07] Per Runeson, Magnus Alexandersson, and Oskar Nyholm. Detection of Duplicate Defect Reports Using Natural Language Processing. In *29th International Conference on Software Engineering*, pages 499–510. IEEE, may 2007.

[RC08] Chanchal K. Roy and James R. Cordy. An Empirical Study of Function Clones in Open Source Software. In *2008 15th Working Conference on Reverse Engineering*, pages 81–90. IEEE, oct 2008.

[RM09] Neha Rungta and Eric G. Mercer. Guided model checking for programs with polymorphism. In *Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation - PEPM '09*, page 21, New York, New York, USA, 2009. ACM Press.

[Roc75] Marc J Rochkind. The source code control system. *Software Engineering, IEEE Transactions on*, (4):364–370, 1975.

[RYR13] Mohammad Masudur Rahman, Shamima Yeasmin, and Chanchal K. Roy. An IDE-based context-aware meta search engine. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 467–471. IEEE, oct 2013.

[RZF+13] Jeremias Rößler, Andreas Zeller, Gordon Fraser, Cristian Zamfir, and George Candea. Reconstructing Core Dumps. In *Proceedings of the 6th International Conference on Software Testing, Verification and Validation, ser. ICST*, 2013.

[SCFP00] John Steven, Pravir Chandra, Bob Fleck, and Andy Podgurski. jRapture: A Capture/Replay Tool for Observation-Based Testing. In *Proceedings of the International Symposium on Software Testing and Analysis.*, number August, pages 158–167, 2000.

[SIK+10] Emad Shihab, Akinori Ihara, Yasutaka Kamei, Walid M. Ibrahim, Masao Ohira, Bram Adams, Ahmed E. Hassan, and Ken-ichi Matsumoto. Predicting Re-opened Bugs: A Case Study on the Eclipse Project. In *2010 17th Working Conference on Reverse Engineering*, pages 249–258. IEEE, oct 2010.

[SK03] R. Subramanyam and M.S. Krishnan. Empirical analysis of CK metrics for object-oriented design complexity: implications for software defects. *IEEE Transactions on Software Engineering*, 29(4):297–310, apr 2003.

[SKP14] Ripon K. Saha, Sarfraz Khurshid, and Dewayne E. Perry. An empirical study of long lived bugs. In *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 144–153. IEEE, feb 2014.

[SLKJ11] Chengnian Sun, David Lo, Siau-Cheng Khoo, and Jing Jiang. Towards more accurate retrieval of duplicate bug reports. *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 253–262, nov 2011.

[SLW+10] Chengnian Sun, David Lo, Xiaoyin Wang, Jing Jiang, and Siau-Cheng Khoo. A discriminative model approach for accurate duplicate bug report retrieval. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10*, volume 1, page 45, New York, New York, USA, may 2010. ACM Press.

[SNH13] Weiyi Shang, Meiyappan Nagappan, and Ahmed E. Hassan. Studying the relationship between logging characteristics and the code quality of platform software. *Empirical Software Engineering*, pages 1–27, 2013.

[The99] The Apache Software Foundation. Log4j 2 Guide - Apache Log4j 2, 1999.

[TLS12] Yuan Tian, David Lo, and Chengnian Sun. Information Retrieval Based Nearest Neighbor Classification for Fine-Grained Bug Severity Prediction. In *2012 19th Working Conference on Reverse Engineering*, pages 215–224. IEEE, oct 2012.

[TNAKN11] Ahmed Tamrawi, Tung Thanh Nguyen, Jafar Al-Kofahi, and Tien N. Nguyen. Fuzzy set-based automatic bug triaging. In *Proceeding of the 33rd international conference on Software engineering - ICSE '11*, page 884, New York, New York, USA, 2011. ACM Press.

[TSL12] Yuan Tian, Chengnian Sun, and David Lo. Improved Duplicate Bug Report Identification. In *2012 16th European Conference on Software Maintenance and Reengineering*, pages 385–390. IEEE, mar 2012.

[VHB+03] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. Model Checking Programs. In *Automated Software Engineering*, volume 10, pages 203–232. Springer, 2003.

[VPK04] Willem Visser, Corina S. Psreanu, and Sarfraz Khurshid. Test input generation with java PathFinder. *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, page 97, 2004.

[Wel13] Brian Wellington. dnsjava, 2013.

[WPZZ07] Cathrin Weiss, Rahul Premraj, Thomas Zimmermann, and Andreas Zeller. How Long Will It Take to Fix This Bug? In *Fourth International Workshop on Mining Software Repositories (MSR'07:ICSE Workshops 2007)*, pages 1–1. IEEE, may 2007.

[WZKC11] Rongxin Wu, Hongyu Zhang, Sunghun Kim, and SC Cheung. Relink: recovering links between bugs and changes. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering.*, pages 15–25, 2011.

[Xst11] Xstream. Xstream, 2011.

[ZGV13] Hongyu Zhang, Liang Gong, and Steve Versteeg. Predicting bug-fixing time: an empirical study of commercial software projects. pages 1042–1051, may 2013.

[ZN08] Thomas Zimmermann and Nachiappan Nagappan. Predicting defects using network analysis on dependency graphs. In *Proceedings of the 13th international conference on Software engineering - ICSE '08*, page 531, New York, New York, USA, may 2008. ACM Press.

[ZNGM12] Thomas Zimmermann, Nachiappan Nagappan, Philip J. Guo, and Brendan Murphy. Characterizing and predicting which bugs get reopened. In *Proceedings of the 34th International Conference on Software Engineering, IEEE*, pages 1074–1083. IEEE, jun 2012.

[ZPZ07] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. Predicting Defects for Eclipse. In *Third International Workshop on Predictor Models in Software Engineering (PROMISE'07: ICSE Workshops 2007)*, pages 9–9. IEEE, may 2007.

[ZZL12] Jian Zhou, Hongyu Zhang, and David Lo. 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*, pages 14–24. IEEE, jun 2012.