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

May 2016

Contents

4

List of Figures

List of Tables

Chapter 1

Introduction

Software maintenance activities such as debugging, refactoring, and feature enhancement are known to be challenging and costly [Pre05]. Studies have shown that the cost of software maintenance can reach up to 70% of the overall cost of the software development life cycle [HR02]. Much of this is attributable to several factors including the increase in software complexity, the lack of traceability between the various artefacts of the software development process, the lack of proper documentation, and the unavailability of the original developers of the systems.

Research in software maintenance has evolved over the years to include areas like mining bug repositories, bug analysis, prevention and reproduction. The ultimate goal is to develop techniques and tools to help software developers detect, correct, and prevent bugs in an effective and efficient manner. Several tools have developed

Despite the recent advances in the field, the literature shows that many existing software maintenance tools have yet to be adopted by industry [LLS$^+$13; FM15; LWA07; APM$^+$07; AP08; JSMHB13; Nor13; HP04a; LvdH11]. This is caused by the following factors:

- Work flow integration: Most of these tools ([RAF04; HP04b; KPW06; AHM$^+$08; Fin15; MGDL10; PNM13; Nay13; NPMG12; NMV13; NBMV15] are some noticeable examples) are not integrated well with the work flow of software developers (i.e. coding, testing, debugging, committing). Indeed, developers are constantly switching from one workspace to another for different tasks (i.e. feature location with a command line tool, development and testing code with an IDE, development and testing front end code with another IDE and a browser,...) [GM04; KMCA06; Lay09; MHJM12] and these tools further interrupt this already fragmented process [RPB$^+$04; RLB06; BKB$^+$06]. Furthermore, these tools do not follow developers across workspace; they are bound to one workspace (i.e. an Eclipse plugin [VHD07]). Hence,

developers have to download, install and understand many tools to achieve any given task.

- Actionable outcome: The outcome of these tools tend do not lead to actionable items. Most of these tools return several results that are hard to interpret. Take for example, FindBugs [HP04a], a well-known bug detection tool. This tool detects 424 bug signatures and reports them using abbreviated code such as `CO_COMPARETO_INCORRECT_FLOATING`. Using this code, developers can browse FindBug's dictionary and find the following definition *"This method compares double or float values using pattern like this: val1 > val2 ? 1 : val1 < val2 ? − 1 : 0. This pattern works incorrectly for -0.0 and NaN [...]"*[Fin15]. While the detection of this bug pattern and the definition are accurate, there is no actionable actions for developers to adjust their code accordingly. Moreover, these tools' output is so verbose that developers tend to simply ignore them [ASWF14; KE07a; KE07b] or use, in addition, filtering and ranking tools [AP10; SFZ11].

- Leverage of historical data: These tools do not leverage a large body of knowledge that already exist in open source systems. In API usage recommendation, researchers built tools that leverage open-source data of multiple projects to recommend how to use a given API [RYR13; KLHK13; MBFV13]. For defect prevention, however, the state-of-the-art approaches consist in adapting statistical models built for a project to another [Lo13; NPK13]. As argued by Lewis *et al.* [LLS$^+$13] and Johnson *et al.* [JSMHB13], approaches based solely on statistical models are perceived as black boxes by developers. Consequently, they are less likely to trust the output of the tool and therefore, use it.

In this thesis, we propose to address the above-mentioned issues by focusing on developing techniques and tools that help software developers at commit-time. In developers's workflow, the commit marks the end of a given task or subtasks as the developer is ready to version the source code. Consequently, actionning historical data, real-world examples in a clear and concise way at commit-time will prevent additional interuptions. Moreover, any real-world project uses a version control system and distilling advices at during the versionning operations, such as commit, will provide an unique experience for developers as they will be able to use the same tool across their different workspaces and projects.

More precisely, we propose the following contributions:

- An aggregated bug repository system.

- A new bug classification based on the locations of the corrections.

- A clone prevention technique at commit-time.

- A bug prevention technique at commit-time

- A bug reproduction technique based on directed model checking and crash traces

When facing a new bug, one might want to leverage decades of open source software history to find a suitable solution. The chances are that a similar bug has already been fixed somewhere in another open source project. The problem is that each open source project hosts its data in a different data repository, using different bug tracking and version control systems. Moreover, these systems have different interfaces to access data. The data is not represented in a uniform way either. This is further complicated by the fact that bug tracking tools and version control systems are not necessarily connected. The former follows the life of the bug, while the latter manages the fixes. As a result, one would have to search the version control system repository to find candidate solutions. Moreover, developers mainly use classical search engines that index specialized sites such as StackOverflow. These sites are organized in the form of question-response where a developer submits a problem and receives answers from the community. While the answers are often accurate and precise, they do not leverage the history of open source software that has been shown to provide useful insights to help with many maintenance activities such as bug fixing [SKP14], bug reproduction [NHLSL15], fault analysis [NAW⁺08], etc.

When a system crashes, software developers need to reproduce the crash (usually in a lab environment) so as to provide corrective measures. A survey conducted with the 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]. Crash reproduction is an expensive task because the data provided by end users is often scarce [AKE08; JO12; Che13a]. It is therefore important to invest in techniques and tools for automatic bug reproduction to ease the maintenance process and accelerate the rate of bug fixes and patches. Existing techniques can be divided into two categories: (a) On-field record and in-house replay [SCFP00; NPC05; AKE08; RNB15], and (b) In-house crash explanation [JO12; JO13; ZJP⁺14; Che13b; NHLSL15].

The first category relies on instrumenting the system in order to capture objects and other system components at run-time. When a faulty behaviour occurs in the field, the stored objects as well as the entire heap are sent to the developers along with the faulty methods to reproduce the crash. These techniques tend to be simple to implement and yield good results, but they suffer from two main limitations. First, code instrumentation comes with a non-negligible overhead on the system. The second limitation is that the

collected objects may contain sensitive information causing customer privacy issues. The second category is composed of tools leveraging proprietary data in order to provide hints on potential causes. While these techniques are efficient in improving our comprehension of the bugs, they are not designed with the purpose of reproducing them.

Code clones appear when developers reuse code with little to no modification to the original code. Studies have shown that clones can account for about 7% to 50% of code in a given software system [Bak; DRD]. Developers often reuse code (and create clones) in their software on purpose [KSN05]. Nevertheless, clones are considered a bad practice in software development since they can introduce new bugs in the code [KG06; JDHW09; LLMZ06]. If a bug is discovered in one segment of the code that has been copied and pasted several times, then the developers will have to remember the places where this segment has been reused in order to fix the bug in each place. In the last two decades, there have been many studies and tools that aim at detecting clones. They can be grouped into three categories. Although these techniques and tools have been shown to be useful in detecting clones, they operate in an off-line fashion (i.e., after the clones have been inserted). Software developers might be reluctant to use these tools on a day-today basis (i.e., as part of the continuous development process), unless they are involved in a major refactoring effort. This problem is somehow similar to the problem of adopting bug identification tools. Johnson et al. [JSMHB13] showed that these tools are challenging to use because they do not integrate well with the day-to-day workflow of a developer. Also they output a large amount of data when applied to the entire system, making it hard to understand and analyse their results.

There have been several studies (e.g., [WZZ07; ZGV13]) that study of the factors that influence the bug fixing time. These studies empirically investigate the relationship between bug report attributes (description, severity, etc.) and the fixing time. Other studies take bug analysis to another level by investigating techniques and tools for bug prediction and reproduction (e.g., [Che13a; KZWZ07; NHLSL15]). These studies, however, treat all bugs as the same. For example, a bug that requires only one fix is analysed the same way as a bug that necessitates multiple fixes. Similarly, if multiple bugs are fixed by modifying the exact same locations in the code, then we should investigate how these bugs are related in order to predict them in the future. Note here that we do not refer to duplicate bugs. Duplicate bugs are marked as duplicate (and not fixed) and only the master bug is fixed.

As a motivating example, consider Bugs #AMQ-5066 and #AMQ-5092 from the Apache Software Foundation bug report management system (used to build one of the datasets in this paper). Bug #AMQ-5066 was reported on February 19, 2014 and solved with a patch provided by the reporter. The solution involves a relatively complex patch that modifies `MQTTProtocolConverter.java`, `MQTTSubscription.java` and `MQTTTest.java` files. The

description of the bug is as follows:

When a client sends a SUBSCRIBE message with the same Topic/Filter as a previous SUBSCRIBE message but a different QoS, the Server MUST discard the older subscription, and resend all retained messages limited to the new Subscription QoS.

A few months later, another bug, Bug #AMQ-5092 was reported:

MQTT protocol converters does not correctly generate unique packet ids for retained and non-retained publish messages sent to clients. [...] Although retained messages published on creation of client subscriptions are copies of retained messages, they must carry a unique packet id when dispatched to clients. ActiveMQ re-uses the retained message's packet id, which makes it difficult to acknowledge these messages when wild card topics are used. ActiveMQ also sends the same non-retained message multiple times for every matching subscription for overlapping subscriptions. These messages also re-use the publisher's message id as the packet id, which breaks client acknowledgement.

This bug was assigned and fixed by a different person than the one who fixed bug #AMQ-5066. The fix consists of modifying slightly the same lines of the code in the exact files used to fix Bug #AMQ-5066. In fact, Bug #5092 could have been avoided altogether if the first developer provided a more comprehensive fix to #AMQ-5066 (a task that is easier said than done). These two bugs are not duplicates since, according to their description, they deal with different types of problems and yet they can be fixed by providing a similar patch. In other words, the failures are different while the root causes (faults in the code) are more or less the same. From the bug handling perspective, if we can develop a way to detect such related bug reports during triaging then we can achieve considerable time saving in the way bug reports are processed, for example, by assigning them to the same developers. We also conjecture that detecting such related bugs can help with other tasks such as bug reproduction. We can reuse the reproduction of an already fixed bug to reproduce an incoming and related bug.

1.1 Research Contributions

In this section, we present our research contributions that are listed here and developed in more detail in the next subsections.

- An aggregate bug repository for developers and Researchers.

- A bug reproduction technique based on a combination of crash traces and model checking.

- An incremental approach for preventing clone insertion at commit time

- A new taxonomy of bugs based on the location of the corrections — an empirical Study

1.1.1 An aggregate bug repository for developers and researchers

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

1.1.2 A bug reproduction technique based on a combination of crash traces and model checking

In this work, we propose an approach, called JCHARMING (Java CrasH Automatic Reproduction by directed Model checkING) that uses a combination of crash traces and model checking to automatically reproduce bugs that caused field failures. Unlike existing techniques, JCHARMING does not require instrumentation of the code. It does not need access to the content of the heap either. Instead, JCHARMING uses a list of functions output when an uncaught exception in Java occurs (i.e., the crash trace) to guide a model checking engine to uncover the statements that caused the crash.

1.1.3 An incremental approach for preventing bug and clone insertion at commit time

In this research, we present PRECINCT (PREventing Clones INsertion at Commit Time) that focuses on preventing the insertion of clones at commit time, i.e., before they reach the central code repository. PRECINCT is an online clone detection technique that relies on the use of pre-commit hooks capabilities of modern source code version control systems. A pre-commit hook is a process that one can implement to receive the latest modification to the source code done by a given developer just before the code reaches the central repository. PRECINCT intercepts this modification and analyses its content to see whether a suspicious clone has been introduced or not. A flag is raised if a code fragment is suspected to be a clone of an existing code segment. In fact, PRECINCT, itself, can be seen as a pre-commit hook that detects clones that might have been inserted in the latest changes with regard to the rest of the source code. This said, only a fraction of the code is analysed, making PRECINCT efficient compared to leading clone detection techniques such as NICAD (Accurate Detection of Near-miss Intentional Clones) [RC08; CR11]. Moreover, the detected clones are presented using a classical diff output that developers are familiar with. PRECINCT is also well integrated with the workflow of the developers since it is used in conjunction with a source code version control systems such as Git.

1.1.4 A new taxonomy of bugs based on the location of the correction — an empirical Study

We investigate the relationship between bugs by examining their locations of the 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.We argue that bugs can be classified into four types: A bug of 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. Type 3 refers to multiple bugs that are fixed in the exact same location. Type 4 is an extension of Type 3, where multiple bugs are resolved by modifying the same set of locations. 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 insights 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).

1.2   Outline

The remaining chapters of this proposal are:

- Chapter 2 - *Background & Related work.* In this chapter, we present the major studies related to our research field, namely, crash reproduction, aggregating bug repositories for mining purposes, and clone detection.

- Chapter 3 - *An Aggregate Bug Repository for Developers and Researchers.* In this chapter, we present `BUMPER` (BUg Metarepository for dEvelopers and Researchers), our bug meta-repository. `BUMPER` acts as our data source for the different contributions.

- Chapter 4 - *JCHARMING: Java CrasH Automatic Reproduction by directed Model checkING.* In this chapter we discuss the components of JCHARMING, the bug reproduction approach we propose.

- Chapter 5 *Preventing Clone Insertion.* This chapter describes one approach to prevent the insertion of clones at commit time.

- Chapter 6 *Preventing Bug Insertion Using Clone Detection.* In this chapter, we present an approach named `BIANCA` (Bug Insertion ANticipation by Clone Analysis at merge time) which uses clone detection to prevent bug insertion.

- Chapter 7 *Remaining Work to Complete the Thesis* presents the remaining work and a publication plan.

Chapter 2

Background & Related Work

In this chapter we present preliminaries in Section 2.1. These preliminaries describe the set of definitions we are using as same as project and versioning systems. Then, we present related works for crash reproduction, crash prediction, linking report and source code modifications and clone detection.

2.1   Preliminaries

Software maintenance, comprehension, evolution, specifications and testing are research areas overlapping each other in terms of terminology.

In this proposal, we will use a precise set of definitions. We do not claim ownership of these definitions, they have been established using various resources [ALRL04; Pra01; Bur06; RGK90; WAC12].

We limit software maintenance to the following three artifacts:

- Bug report: A bug report describes a behavior observed in the field and considered abnormal by the reporter. Bug reports are submitted manually to bug report systems (bugzilla/jira). There is no mandatory format to report a bug, nevertheless, it should have: Version of the software / OS / Platform used, steps to reproduce, screen shots, stack trace and anything that could help a developer to assess the internal state of the software system.

- Crash report: A crash report is the last action a software system does before crashing. Crash reports are automatic (they have to be implemented into the software system by developer) and contain data (that can be proprietary) to help developers understand the crash (e.g. memory dump,...).

- Tasks: A task is a new feature, or the improvement of an existing feature, to be implemented in a future release of the software.

These artifacts are produced in response to the following phenomena:

- Software Bug: A software bug is an error, flaw, failure, defect or fault in a computer program or system that causes it to violate at least one of its functional or nonfunctional requirements.

- Error: An error is a mistake, misconception, or misunderstanding on the part of a software developer.

- Fault/defect: A fault (defect) is defined as an abnormal condition or defect at the component, equipment, or subsystem level which may lead to a failure. A fault (defect) is not final (the system still works) and does not prevent a given feature to be accomplished. A fault (defect) is a deviation (anomaly) of the healthy system that can be caused by an error or external factors (hardware, third parties, ...).

- Failure: The inability of a software system or component to perform its required functions within specified requirements.

- Crash: The software system encountered a fault (defect) that triggered a fatal failure from which the system could not recover from/overcome. As a result, the system stopped.

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

2.1.1 Version control systems

Version control consists of maintaining the versions of files — such as source code and other software artifacts [Zel97]. This activity is a complex task and cannot be performed manually on real world project. Consequently, numerous tools have been created to help practitioners manage the version of their software artifacts. Each evolution of a software is a version (or revision) and each version (revision) is linked to the one before through modifications of software artifacts. These modifications consist of updating, adding or deleting software artifacts. They can be referred as `diff`, `patch` or `commit`[1]. Each `diff`, `patch` or `commit` have the following characteristics:

---

[1]These names are not to be used interchangeably as difference exists.

- Number of Files: The number of software files that have been modified, added or deleted.

- Number of Hunks: The number of consecutive code blocks of modified, added or deleted lines in textual files. Hunks are used to determine, in each file, how many different places the developer has modified.

- Number of Churns: The number of lines modified. However, the churn value for a line change should be at least two as the line has to be deleted first and then added back with the modifications.

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

Branching have been used for more than testing hazardous refactoring or testing framework upgrades. Indeed, task branching is an agile branching strategy where a new branch is created for each task [Mar09]. It is common to see branch named `123_implement_X` where `123` is the `#id` of task `X` given by the project tracking system. Project tracking systems are presented in Section 2.1.2.

In modern versioning systems, when maintainers make modifications to the source code want to version it, they have to do commit. The commit operation will version the modifications applied to one or many files.

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

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

Branches point to a commit. In a task-branching environment, a branch is created via a checkout operation for each task. Tasks can be to fix the root cause of a crash or bug
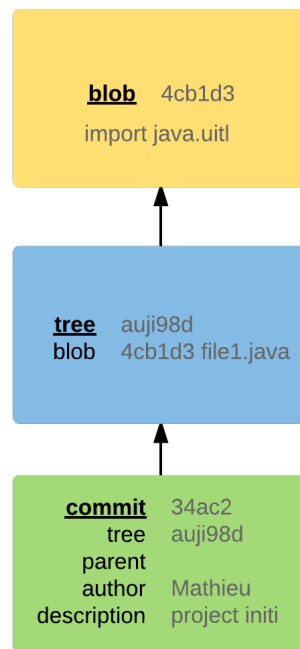
Figure 1: Data structure of a commit.
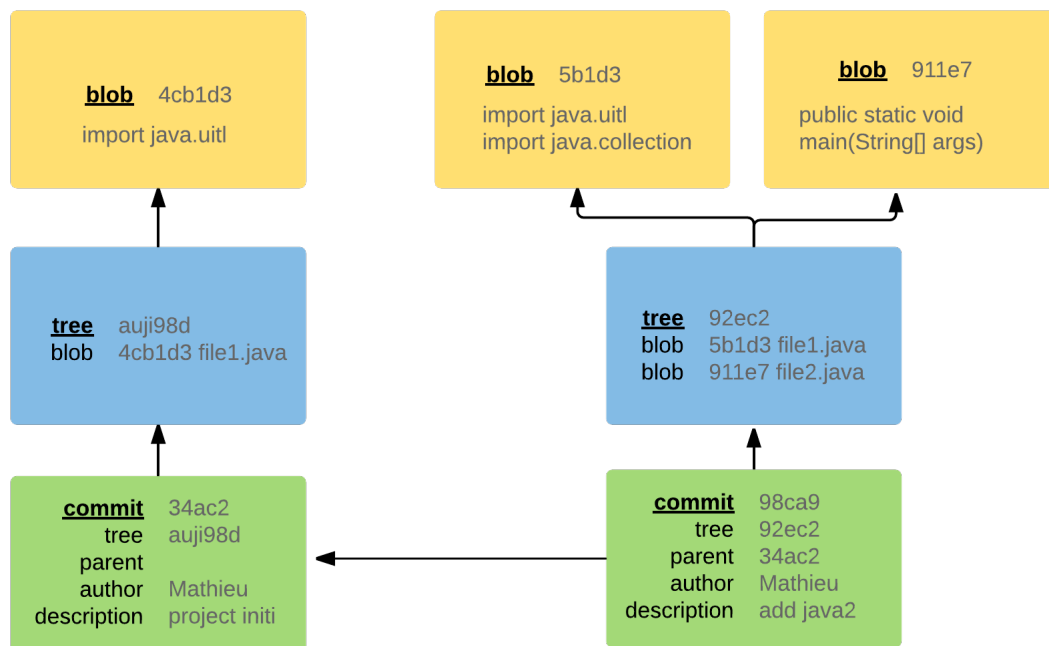
Figure 2: Data structure of two commits.

report or features to implement. In figure 3, the *master* branch and the $1\_fix\_overflow$ point on commit $98ca9$.
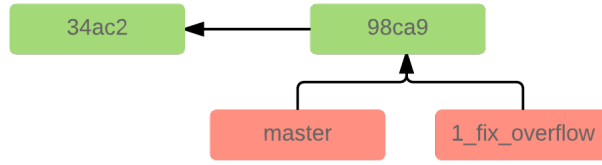
Figure 3: Two branches pointing on one commit.

Both branches can evolve separately and be merged together when the task branch is ready. In figure 4, the $master$ branch points on $a13ab2$ while the $1\_fix\_overflow$ points on $ahj23k$.
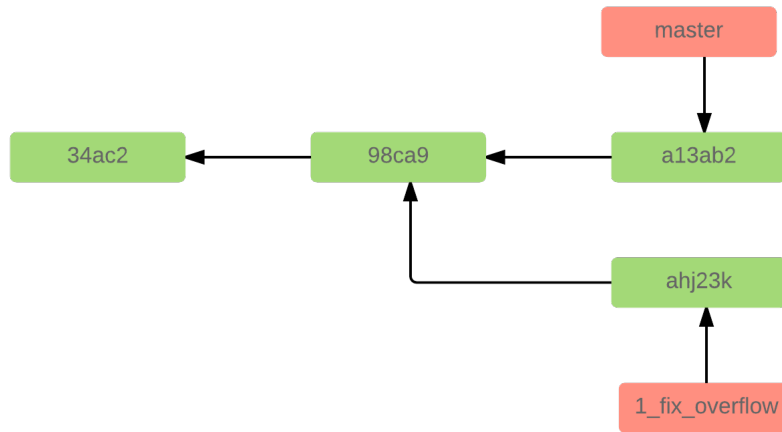
Figure 4: Two branches pointing on two commits.

Providers

In this proposal, we mainly refer to three version control systems: `Svn`, `Git` and, to a lesser extent, `Mercurial`. SVN is distributed by the Apache foundation and is a centralized concurrent version system that can handle 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`.

13

2.1.2 Project Tracking Systems

Project tracking systems allow end users to create bug reports (BRs) to report unexpected system behavior, manager can create tasks to drive the evolution forward and crash report (CRs) can be automatically created. These systems are also used by development teams to keep track of the modification induced by bug and to crash reports, and keep track of the fixes.
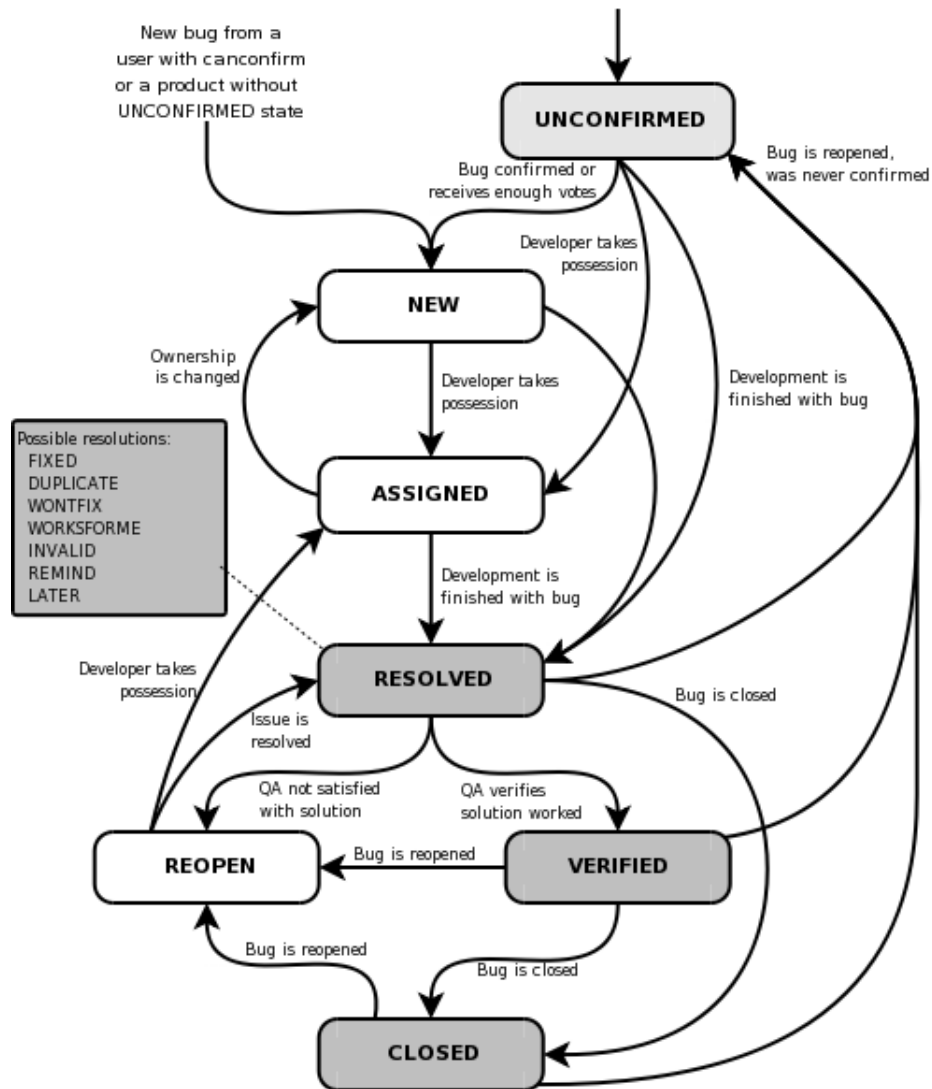
Figure 5: Lifecyle of a report [Bug08]

Figure 5 presents the life cycle of a report. When a report 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 report is then assigned to a developer to

be fixed. When the report is in the `ASSIGNED` state, the assigned developer(s) starts working on the report. A fixed report moves to the `RESOLVED` state. Developers have five different possibilities to resolve a report: `FIXED`, `DUPLICATE`, `WONTFIX`, `WORKSFORME` and `INVALID` [Kop06].

- `RESOLVED/FIXED`: A modification to the source code has been pushed, i.e., a changeset (also called a patch) has been committed to the source code management system and fixes the root problem described in the report.

- `RESOLVED/DUPLICATE`: A previously submitted report is being processed. The report is marked as duplicate of the original report.

- `RESOLVED/WONTFIX`: This is applied in the case where developers decide that a given report will not be fixed.

- `RESOLVED/WORKSFORME`: If the root problem described in the report cannot be reproduced on the reported OS / hardware.

- `RESOLVED/INVALID`: If the report is not related to the software itself.

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

Tasks follow a similar life cycle with the exception of the `UNCONFIRMED` and `RESOLVED` states. Tasks are created by management and do not need to be confirmed 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. Bug and crash reports are considered as problems to eradicate in the program. Tasks are considered as new features or amelioration to include in the program.

Reports and tasks can have a severity[BJS+08]. The severity is a classification to indicate the degree of impact on the software. The possible severities are:

- blocker: blocks development and/or testing work.

- critical: crashes, loss of data, severe memory leak.

- major: major loss of function.

- normal: regular report, some loss of functionality under specific circumstances.

- minor: minor loss of function, or other problem where easy workaround is present.

- trivial: cosmetic problems like misspelled words or misaligned text.

The relationship between an report or a task and the actual modification can be hard to establish and it has been a subject of various research studies (e.g., [ACC+02; BBR+10; WZKC11]). This reason is that they are in two different systems: the version control system and the project tracking system. While it is considered a good practice to link each report with the versioning system by indicating the report $\#id$ on the modification message, more than half of the reports are not linked to a modification[WZKC11].

Providers

We have collected and plan to collect data from four different project tracking systems: *Bugzilla*, *Jira*, *Github* and *Sourceforge*. *Bugzilla* belongs to the Mozilla foundation and has first been released in 1998. *Jira*, provided by 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 source code revision system and evolve, later on, to add project tracking capabilities to their softwares. This common particularity have the advantage to ease the link between reports and source code.

2.2   Crash reproduction

The first (and perhaps main) step in understanding the cause of a field crash is to reproduce the bug that caused the system to fail. A survey conducted with developers of major open source software systems such as Apache, Mozilla and Eclipse revealed that one of the most valuable piece of information that can help locate and fix the cause of a crash is the one that can help reproduce it [BJS+08].

Crash reproduction is, however, a challenging task because of the limited amount of information provided by the end users. There exist several bug reproduction techniques. They can be grouped into two categories: (a) On-field record and in-house replay [NPC05; AKE08; JKXC10], and (b) In-house crash explanation [MSA04; CFS09]. The first category relies on instrumenting the system in order to capture objects and other system components at run-time. When a faulty behavior occurs in the field, the stored objects, as well as the entire heap, are sent to the developers along with the faulty methods to reproduce the crash. These techniques tend to be simple to implement and yield good results, but they suffer from two main limitations. First, code instrumentation comes with a non-negligible overhead on the system. The second limitation is that the collected objects may contain sensitive information causing customer privacy issues. The second category is composed of tools leveraging proprietary data in order to provide hints on potential causes. While these techniques are efficient in improving our comprehension of the bugs, they are not designed with the purpose of reproducing them.

These two categories yield varying results depending on the selected approach and are mainly differentiated by the need for instrumentation. The first category of techniques oversees — by means of instrumentation — the execution of the target system on the field in order to reproduce the crashes in-house, whereas tools and approaches belonging to the second category only use data produced by the crash such as the crash stack or the core dump at crash time. In the first category, tools record different types of data such as the invoked methods [NPC05], try-catch exceptions [RZF$^+$13], or objects [JKXC10]. In the second category, existing tools and approaches are aimed towards understanding the causes of a crash, using data produced by the crash itself, such as a crash stack [Che13b], previous — and controlled — execution [ZJP$^+$14], etc.

Tools and approaches that rely on instrumentation face common limitations such as the need to instrument the source code in order to introduce logging mechanisms[NPC05; JKXC10; AKE08], which is known to slow down the subject system. In addition, recording system behavior by means of instrumentation may yield privacy concerns. Tools and approaches that only use data about a crash — such as core dump or exception stack crashes — face a different set of limitations. They have to reconstruct the timeline of events that have led to the crash [Che13b; NHLSL15]. Computing all the paths from the initial state of the software to the crash point is an NP-complete problem, and may cause state space explosion [Che13b; CO07].

In order to overcome these limitations, some researchers have proposed to use various SMT (satisfiability modulo theories) solvers [DM06] and model checking techniques

[VHB$^+$03]. However, these techniques require knowledge that goes beyond traditional software engineering, which hinders their adoption [VPK04].

It is worth mentioning that both categories share a common limitation. It is possible for the required condition to reproduce a crash to be purely external such as the reading of a file that is only present on the hard drive of the customer or the reception of a faulty network packet [Che13b; NHLSL15]. It is almost impossible to reproduce the bug without this input.

On-field Record and In-house Replay

Jaygarl *et al.* created OCAT (Object Capture based Automated Testing) [JKXC10]. The authors' approach starts by capturing objects created by the program when it runs on-field in order to provide them to an automated test process. The coverage of automated tests is often low due to lack of correctly constructed objects. Also, the objects can be mutated by means of evolutionary algorithms. These mutations target primitive fields in order to create even more objects and, therefore, improve the code coverage. While not directly targeting the reproduction of a bug, OCAT is an approach that was used as the main mechanism for bug reproduction systems.

Narayanasamy *et al.* [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.

Another approach in this category was proposed by Clause *et al.* [CO07]. The approach records the execution of the program on the client side and compresses the generated data. Moreover, the approach keeps compressed traces of all accessed documents in the operating system. This data is sent to the developers to replay the execution of the program in a sandbox, simulating the client's environment. This special feature of the approach proposed by Clause *et al.* addresses the limitation where crashes are caused by external causes. While the authors broaden the scope of reproducible bugs, their approach records a lot of data that may be deemed private such as files used for the proper operation of the operating system.

Timelapse [BBKE13] also addresses the problem of reproducing bugs using external data. The tool focuses on web applications and allows developers to browse and visualize the execution traces recorded by Dolos. Dolos captures and reuses user inputs and network

responses to deterministically replay a field crash. Also, both Timelapse and Dolos allow developers to use conventional tools such as breakpoints and classical debuggers. Similar to the approach proposed by Clause *et al.* *[CO07]*, private data are recorded without obfuscation of any sort.

Another approach was proposed by Artzi *et al.* and named ReCrash. ReCrash records the object states of the targeted programs [AKE08]. The authors use an in-memory stack, which contains every argument and object clone of the real execution in order to reproduce a crash via the automatic generation of unit test cases. Unit test cases are used to provide hints to the developers about the buggy code. This approach particularly suffers from the limitation related to slowing down the execution. The overhead for full monitoring is considerably high (between 13% and 64% in some cases). The authors propose an alternative solution in which they record only the methods surrounding the crash. For this to work, the crash has to occur at least once so they could use the information causing the crash to identify the methods surrounding it. ReCrash was able to reproduce 100% (11/11) of the submitted bugs.

Similar to ReCrash, JRapture [SCFP00] is a capture/replay tool for observation-based testing. The tool captures the execution of Java programs to replay it in-house. To capture the execution of a Java program, the creators of JRapture used their own version of the Java Virtual Machine (JVM) and a lightweight, transparent capture process. Using a customized JVM allows capturing any interactions between a Java program and the system including GUI, files, and console inputs. These interactions can be replayed later with exactly the same input sequence as seen during the capture phase. However, using a custom JVM is not a practical solution. This is because, the authors' approach requires from users to install a JVM that might have some discrepancies with the original one and yield bugs if used with other software applications. In our view, JRapture fails to address the limitations caused by instrumentation because it imposes the installation of another JVM that can also monitor other software systems than the intended ones. RECORE (REconstructing CORE dumps) is a tool proposed by Robler *et al.*. The tool instruments Java byte code to wrap every method in a try-catch block while keeping a quasi-null overhead [RZF$^+$13]. RECORE starts from the core dump and tries (with evolutionary algorithms) to reproduce the same dump by executing the subject program many times. When the generated dump matches the collected one, the approach has found the set of inputs responsible for the failure and was able to reproduce 85% (6/7) of the submitted bugs.

The approaches presented at this point operate at the code level. There exist also techniques that focus on recording user-GUI interactions [HGWB11; RNB15]. Roehm *et al.* extract the recorded data using delta debugging [ZH02], sequential pattern mining, and

their combination to reproduce between 75% and 90% of the submitted bugs while pruning 93% of the actions.

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

On-house Crash Explanation

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

Jin *et al.* proposed BugRedux for reproducing field failures for in-house debugging [JO12]. The tool aims to synthesize in-house executions that mimic field failures. To do so, the authors use several types of data collected in the field such as stack traces, crash stack at points of failure, and call sequences. The data that successfully reproduced the field crash is sent to software developers to fix the bug. BugRedux relies on several in-house executions that are synthesized so as to narrow down the search scope, find the crash location, and finally reproduce the bug. However, these in-house executions have to be conducted before the work on the bug really begins. Also, the in-house executions suffer from the same limitation as unit testing, *i.e.*, the executions are based on the developer's knowledge and ability to develop exceptional scenarios in addition to the normal ones. Based on the success of BugRedux, the authors built F3 (Fault localization for Field Failures) [JO13] and MIMIC [ZJP+14]. 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. MIMIC further extends F3 by comparing a model of correct behavior to failing executions and identifying violations of the model as potential explanations for failures.

Likewise, Zamfir *et al.* proposed ESD [ZC10], an execution synthesis approach that automatically synthesizes failure execution using only the stack trace information. However,

this stack trace is extracted from the core dump and may not always contain the components that caused the crash.

To the best of our knowledge, the most complete work in this category is the one of Chen in his Ph.D thesis [Che13b]. Chen proposed an approach named STAR (Stack Trace based Automatic crash Reproduction). Using only the crash stack, STAR starts from the crash line and goes backward towards the entry point of the program. During the backward process, STAR computes the required condition using an SMT solver named Yices [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. STAR aims to tackle the state explosion problem of reproducing a bug by reconstructing the events in a backward fashion and therefore saving numerous states to explore. STAR was able to reproduce 38 crashes out of 64 (54.6%). Also, STAR is relatively easy to implement as it uses Yices [DM06] and potentially Z3 [DB08] (stated in their future work) that are well-supported SMT solvers.

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

2.3 Reports and source code relationships

Mining bug repositories is perhaps one of the most active research fields today. The reason is that the analysis of bug reports (BRs) provides useful insight that can help with many maintenance activities such as bug fixing [WZZ07; SKP14] bug reproduction [Che13a; AKE08; JO12], fault analysis [NAW+08], etc. This increase of attention can be further justified by the emergence of many open source bug tracking systems, allowing software teams to make their bug reports available online to researchers.

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

Researchers have been studying the relationships between the bug and source code repositories since more than two decades. To the best of our knowledge the first ones who conducted this type of study on a significant scale were Perry and Stieg [PS93]. In these

two decades, many aspects of these relationships have been studied in length. For example, researchers were interested in improving the bug reports themselves by proposing guidelines [BJS+08], and by further simplifying existing bug reporting models [HGGBR08].

Another field of study consist of assigning these bug reports, automatically if possible, to the right developers during triaging [AHM06; JKZ09; TNAKN11; BvdH13]. Another set of approaches focus on how long it takes to fix a bug [BN11; ZGV13; SKP14] and where it should be fixed [Zel13; ZZL12]. With the rapidly increasing number of bugs, the community was also interested in prioritizing bug reports [KWM+11], and in predicting the severity of a bug [LDGG10]. Finally, researchers proposed approaches to predict which bug will get reopened [ZNGM12; Lo13], which bug report is a duplicate of another one [BPZ08; TSL12; JW08] and which locations are likely to yield new bugs [KZPJ06; KZWZ07].

In her PhD thesis [Eld01], Sigrid Eldh discussed the classification of trouble reports with respect to a set of fault classes that she identified. Fault classes include computational logical faults, ressource faults, function faults, etc. She conducted studies on Ericsson systems and showed the distributions of trouble reports with respect to these fault classes. A research paper was published on the topic in [EPHJ07]. Hamill et al.[HGP14] proposed a classification of faults and failures in critical safety systems. They proposed several types of faults and show how failures in critical safety systems relate to these classes. They found that only a few fault types were responsible for the majority of failures. They also compare on pre-release and post-release faults and showed that the distributions of fault types differed for pre-release and post-release failures. Another finding is that coding faults are the most predominant ones.

2.4   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 [KZWZ07]. 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 [MPNC12]. 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; Nay13], 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.

2.5   Clone Detection

Some of our contributions rely on code clone detection to perform their functionalities. Consequently, we reviewed the litterature of the field. These section describes major works in clone detection.

Code clones appear when developers reuse code with little to no modification to the original code. Studies have shown that clones can account for about 7% to 50% of code in a given software system[Bak; DRD]. Developers often reuse code (and create clones) in their software on purpose[KSN05]. Nevertheless, clones are considered a bad practice in software development since they can introduce new bugs in the code[KG06; JDHW09; LLMZ06]. If a bug is discovered in one segment of the code that has been copied and pasted several times, then the developers will have to remember the places where this segment has been reused in order to fix the bug in each place.

In the last two decades, there have been many studies and tools that aim at detecting clones. They can be grouped into three categories. The first category includes techniques that treat the source code as text and use transformation and normalization methods to compare various code fragments[Joh94; Joh93; CR11; RC08]. The second category includes methods that use lexical analysis, where the source code is sliced into sequences of tokens, similar to the way a compiler operates[Bak; Bak92; BG02; KKI02a; LLMZ06]. The tokens are used to compare code fragments. Finally, syntactic analysis has also been performed where the source code is converted into trees, more particularly abstract syntax tree (AST), and then the clone detection is performed using tree matching algorithms[BYM+98; KH00; TG06; FFK08].

Although these techniques and tools have been shown to be useful in detecting clones, they operate in an offline fashion (i.e., after the clones have been inserted). Software developers might be reluctant to use these tools on a day-to-day basis (i.e., as part of the continuous development process), unless they are involved in a major refactoring effort. Johnson et al. [JSMHB13] showed that these tools are challenging to use because they do not integrate well with the day-to-day workflow of a developer. Also they output a large amount of data when applied to the entire system, making it hard to understand and analyse their results.

Text-based techniques use the code — often raw (e.g. with comments) — and compare

sequences of code (blocks) to each other in order to identify potential clones. Johnson was perhaps the first one to use fingerprints to detect clones[Joh93; Joh94]. Blocks of code are hashed, producing fingerprints that can be compared. If two blocks share the same fingerprint, they are considered as clones. Manber et al. [Man94] and Ducasse et al.[DRD99] refined the fingerprint technique by using leading keywords and dot-plots.

Tree-matching and metric-based are two sub-categories of syntactic analysis for clone detection. Syntactic analysis consists of building abstract syntax trees (AST) and analyse them with a set of dedicated metrics or searching for identical sub-trees. Many approaches using AST have been published using sub-tree comparison including the work of Baxter et al.[BYM$^+$98], Wahleret et al. [WSWF], or more recently, the work of Jian et al. with Deckard [JMSG07]. An AST-based approach compares metrics computed on the AST, rather than the code itself, to identify clones [PMDL99; BMD$^+$].

Another approach to detect clones is to use static analysis and to leverage the semantics of the program to improve the detection. These techniques rely on program dependency graphs where nodes are statements and edges are dependencies. Then, the problem of finding clones is reduced to the problem of finding identical sub-groups in the program dependency graph. Examples of recent techniques that fall into this category are the ones presented by Krinke et al.[Kri01] and Gabel et al. [GJS08].

Many clone detection tools have been created using a lexical approach for clone detection. Here, the code is transformed into a series of tokens. If sub-series repeat themselves, it means that a potential clone is in the code. Some popular tools that use this technique include, but not limited to, Dup[Bak], CCFinder[KKI02a], and CP-Miner[LLMZ06].

Furthermore, a large number of taxonomies have been published in an attempt to classify clones and ease the research on clone detection[MLM96; BMD$^+$99; KFF06; BKA$^+$07; Kon; KG].

Other active research activities in clone detection focus on clone removal and management. Once detected, an obvious step is to provide approaches to remove clones in an automatic way or (at least) keep track of them if removing them is not an option. Most modern IDEs provide the *extract method* feature that transforms a potentially copy-pasted block of code into a method and a call to the newly generated method[KH; HKKI04]. More advanced techniques (see Codelink[TBG] and[DER07]) involve analysing the output of CCFinder[KKI02b; LHMI07] or program dependencies graphs[HKKI04] to automatically suggest a method that would go through the *extract method* process.

The aforementioned techniques, however, focus on detecting clones after they are inserted in the code. Only a few studies focus on preventing the insertion of clones. Lague et al. [LPM$^+$] conducted a very large empirical study with 10,000 developers over 3 years,

where developers where asked to use clone detection tools during the development process of a very large telecom system. The authors found that while clones are being removed over time, using clone detection tools help improving the quality of the system as it prevents defects to reach the customers. Duala et al. [DER07; DER10] proposed to create clone region descriptors (CRDs), which describe clone regions within methods in a robust way that is independent from the exact text of the clone region or its location in a file. Then, using CRDs, clone insertion can be prevented.

Finally, many taxonomies have been published in an attempt to classify clones into types. [MLM96; BMD$^+$99; KFF06; BKA$^+$07; Kon; KG]. Despite the particularities of each proposed taxonomy, researchers agree on the following classification. Type 1 clones are copy-pasted blocks of code that only differ from each other in terms of non-code artifacts such as indentation, whitespaces, comments and so on. Type 2 clones are blocks of code that are syntactically identical at the exception of literals, identifiers and types that can be modified. In addition, Type 2 clones share the particularities of Type 1 about indentation, whitespaces and comments. Type 3 clones are similar to Type 2 clones in terms of modification of literals, identifiers, types, indentation, whitespaces and comments but also contain added or deleted code statements. Finally, Type 4 are code blocks that perform the same tasks, but using a completely different implementation.

Chapter 3

An Aggregate Bug Repository for Developers and Researchers

In this chapter, we present `BUMPER` (BUg Metarepository for dEvelopers and Researchers). The role of `BUMPER` is to aggregate information belonging to the version control and project management systems. `BUMPER` acts as our consolidated dataset.

The materials presented in this chapter are based on the following publications:

- Nayrolles, M. & Hamou-Lhadj, W. BUMPER: A Tool to Cope with Natural Language Search of Millions Bugs and Fixes. In Proceeding of the International Conference on Software Analysis, Evolution, and Reengineering (SANER'16) - Tool Track, pages 649-652, 2016.

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

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 https://bumper-app.com, (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.

3.1   Data collection

Figure 8 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 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.
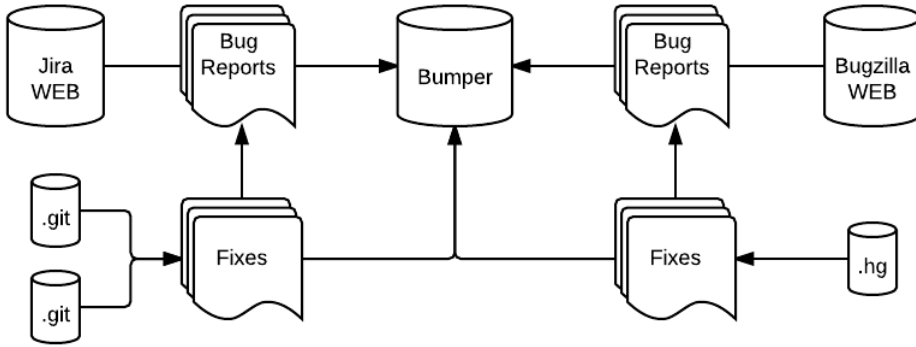
Figure 6: Overview of the bumper database construction.

We used the same datasets presented earlier in Table **??**. 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.

---

[1]https://netbeans.org/bugzilla/
[2]https://issues.apache.org/jira/issues/?jql=
[3] http://mercurial.selenic.com/
[4]http://git-scm.com/
[5]https://github.com/apache

29

3.2 Architecture

BUMPER rely on a highly scalable architecture composed of two distinct servers as depicted in Figure 7. 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.

- 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.

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.
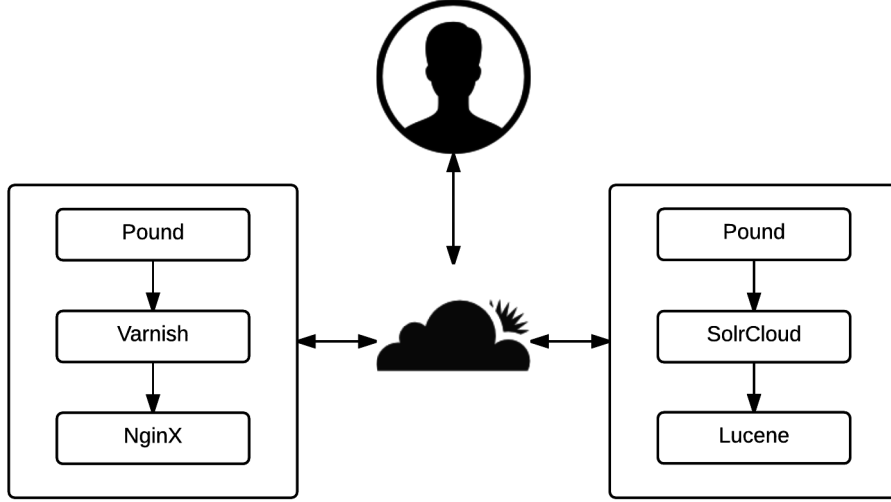
Figure 7: Overview of the bumper architecture.

3.3 UML Metamodel

Figure 8 presents the simplified `BUMPER` metamodel that we designed according to our bug taxonomy presented in section 26 and according to our future needs for `JCHARMING`, `RESSEMBLE` and `BIANCA`.

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*.

3.4 Features

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

3.4.1 Bug Report

A bug report is characterized by the following features:

- ID: unique string id of the form bug_dataset_project_bug_id
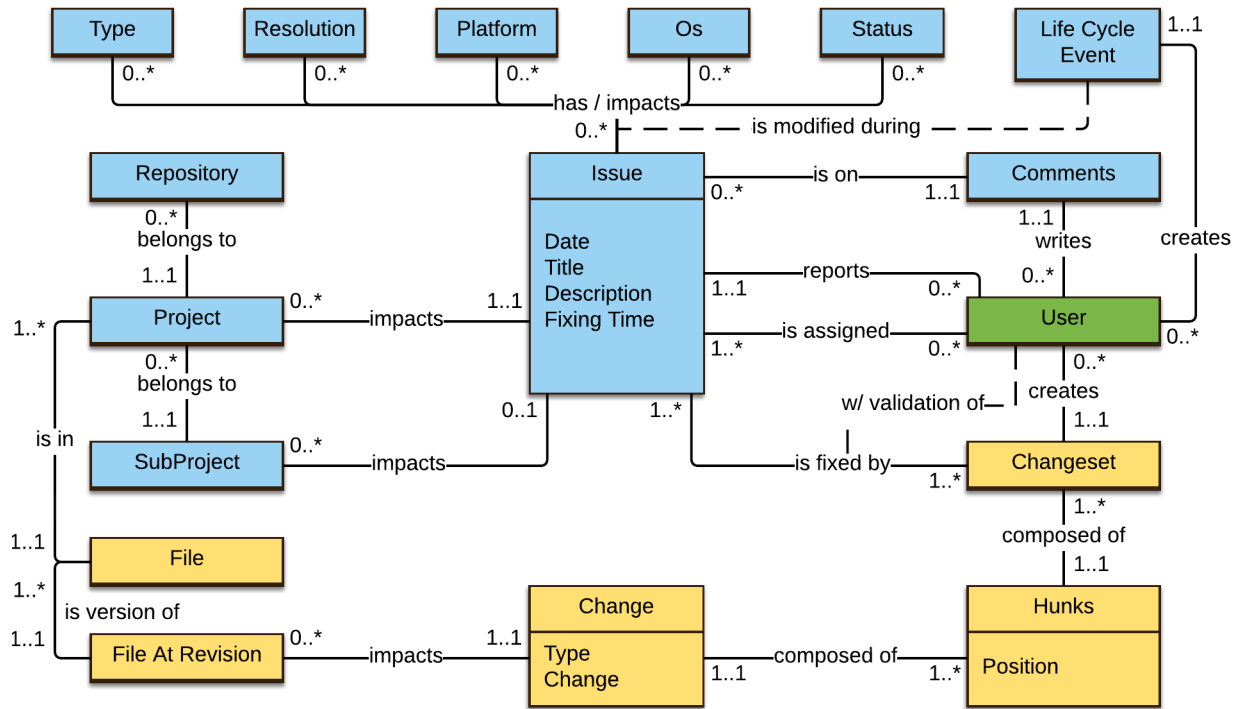
Figure 8: Overview of the bumper meta-model.

- 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 .

3.4.2 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.

3.4.3   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.

3.5   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:

$$(type : BUG\ AND\ report\_t : (``YOUR\ TERMS'')) \ OR\ (!parent\ which = type``BUG'')$$
$$fix\_t : ``YOUR\ TERMS'')$$

(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
($!parent\ which = type``BUG'')fix\_t : ``YOUR\ TERMS''$). 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 9 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.

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 chapters.

---

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

Figure 9: Screenshot of https://bumper-app.com with "Exception" as research.

Chapter 4

JCHARMING: Java CrasH Automatic Reproduction by directed Model checkING

In this chapter, we present JCHARMING (Java CrasH Automatic Reproduction by directed Model checkING). JCHARMING is an approach to reproduce field-crash. Every time a bug report is submitted to a project management system and aggregated by BUMPER, JCHARMING will try to reproduce it. In case of success, the steps to reproduce the bug are saved in BUMPER.

The materials presented in this chapter are based on the following publications:

- 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. 2016. (Accepted).

- Nayrolles, M. , Hamou-Lhadj, W., Tahar, S. Larsson, A. JCHARMING : A Bug Reproduction Approach Using Crash Traces and Directed Model Checking. In Proceeding of the International Conference on Software Analysis, Evolution, and Reengineering (SANER'15), pages 101-110, 2015. (Best Paper Award).

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.

JCHARMING's directed model checking overcomes the state explosion problem of classical model checking techniques and allows the generation of JUnit test cases in a reasonable amount of time. JCHARMING is also easy to deploy. It does not require instrumentation, and hence does not require access to data that may potentially be considered confidential. Moreover, JCHARMING offers better results than approaches described in Section 2.2 that only use bug report data. To assess the efficiency of `JCHARMING` we try to reproduce bug reports contained in `BUMPER` and uur approach is able to reproduce 80% (24/30) of bugs. Moreover, it outperforms STAR (54.6%) [Che13a] and BugRedux (37.5%) [JO12].

4.1   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:

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

38

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 10, 11 and 12 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 10 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 this knowledge in JCHARMING.

Model checking, on the other hand, explores each and every state of the program (Figure
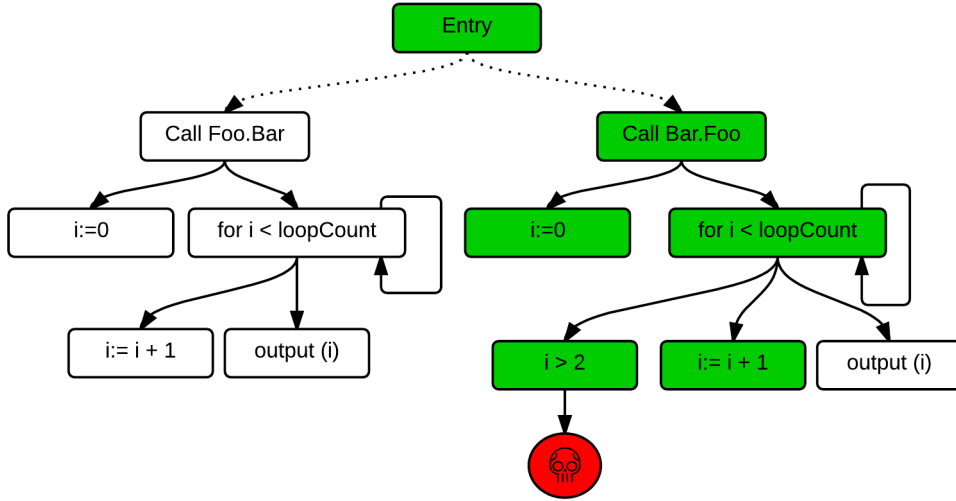
Figure 10: A toy program under testing
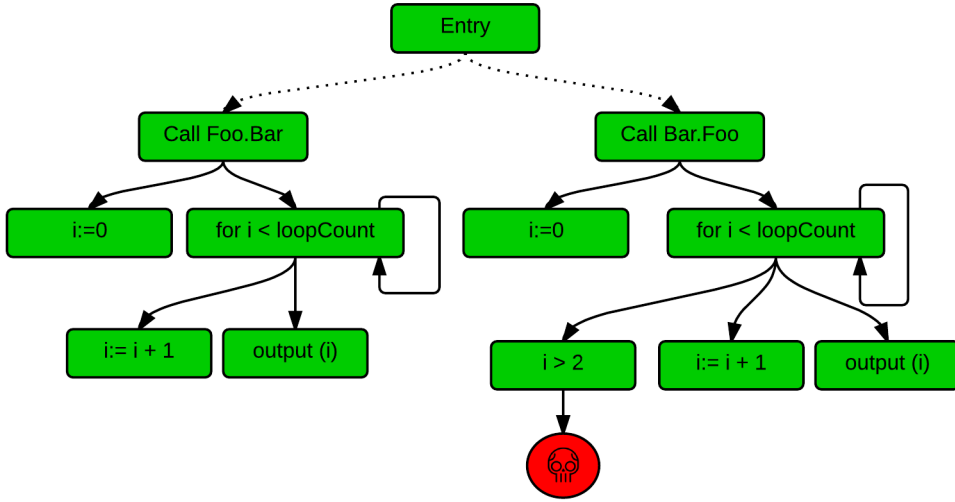
Figure 11: A toy program under model checking

11), 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 12 explores only the states that may lead to a specific location, in our case, the location of the fault. The challenge, however, is to design techniques that can guide the model checking engine. As we will describe in the next section, we use crash traces and program slicing to overcome
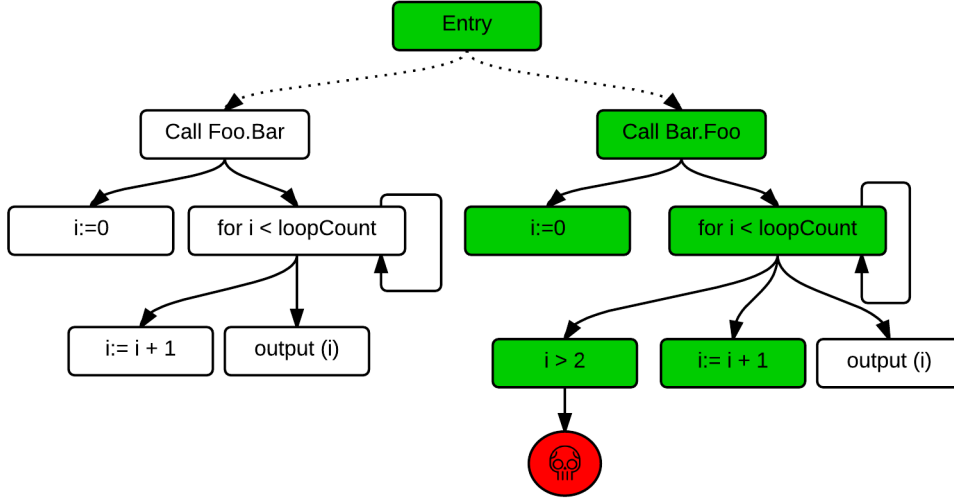
Figure 12: A toy program under directed model checking

this challenge.

4.2 The JCHARMING Approach

Figure 13 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.

4.2.1 Collecting Crash Traces

The first step of JCHARMING is to collect the crash trace caused by an uncaught exception. Crash traces are usually included in crash reports and can therefore be automatically retrieved using a simple regular expression. Figure 14 shows an example of a crash trace

41
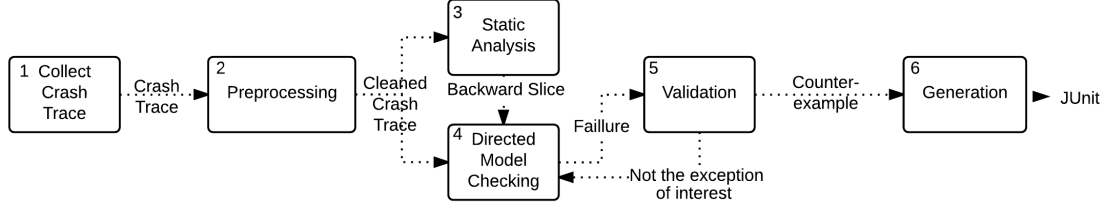
Figure 13: Overview of JCHARMING.

that contains the exception thrown when executing the program depicted in Figures 10 to 12. 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 14: Java InvalidActivityException is thrown in the Bar.Goo loop if the control variable is greater than 2.

As shown in Figure 14, 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 14 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."*

42

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

4.2.2 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 14, 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.

4.2.3 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 14, 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:

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

Indeed, in Figure 15, 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 $\cup_{i=0}^{n} bslice_{[f_{i+1} \leftarrow f_i]}$.
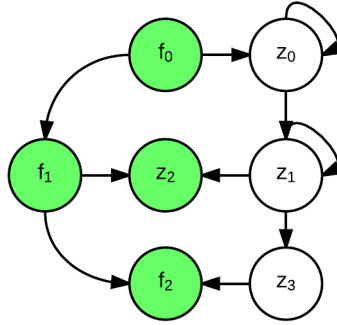
Figure 15: Hypothetical example representing $bslice_{[entry \leftarrow f_0]}$ Vs. $\cup_{i=0}^{n} bslice_{[f_{i+1} \leftarrow f_i]} = \{f_0, f_1, f_2, z_2\}$

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* $\leftarrow$ *extract frames from crash stack*;
>
> Int n $\leftarrow$ size of frame;
>
> Int offset $\leftarrow$ 1;
>
> Bslice bSlice $\leftarrow$ $\emptyset$;
>
> **for** $i \leftarrow$ *0 to i < n && offset < n - 1* **do**
>
> > BSlice currentBSlice $\leftarrow$ backward slice from frames[i] to i + offset;
> >
> > **if** *currentBSlice* $\neq \emptyset$ **then**
> >
> > > bSlice $\leftarrow$ bSlice $\cup$ currentBSlice;
> > >
> > > offset $\leftarrow$ 1;
> >
> > **else**
> >
> > > offset $\leftarrow$ 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 10 to 12 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.

4.2.4   Directed Model Checking

The model checking engine we use in this section 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 $x.\bigcup_{i=0}^{entry} bslice_{[f_{i+1} \leftarrow f_i]} \subset x.SUT$.

4.2.5 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 crash 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 section, we consider a crash 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.

4.2.6 Generating Test Cases for Bug Reproduction

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

4.3   Case studies

In this section, we show the effectiveness of JCHARMING to reproduce bugs in seven open source systems[1] . 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?*

4.3.1   Targeted Systems

Table 1 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 1: 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.

---

[1]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.

4.3.2   Bug Selection and Crash Traces

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

4.4   Results

Table 2 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 2: Effectiveness of JCHARMING using directed model checking (DMC) and model checking (MC) in minutes

As we can see in Table 2, 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.

4.4.1 Successfully reproduced

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

The reporter also attached the following crash trace that we used as input for JCHARM-ING:

```
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 Multicaster.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.

51

4.4.2 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 Chart-MouseEvent'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.

4.4.3 Not Reproduced

To conclude the discussion on the case study, we present a case where JCHARMING was unable to reproduce the failure. For the bug #47957 belonging to Log4j and reported in late 2009 the 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.

Chapter 5

Preventing Clone Insertion

The adoption of tools and processes that aim to support human maintainers during maintenance is relatively low [LLS⁺13; FM15; LWA07; APM⁺07; AP08; JSMHB13; Nor13; LvdH11]. Human maintainers agree that such tools are beneficial, but, in addition of disrupting their workflow, these tools tend to have a high false positive rate. Also, the way in which the warnings are presented, among other things, are barriers to use[JSMHB13]. When asked for their opinions, human maintainers agree to the following characteristics for maintenance-oriented tools[HP04a; LvdH11; LLS⁺13]:

- Actionable messages. Presenting a warning about bug-proneness of a given line is not enough. Clear actions to improve the source code should be provided

- Obvious reasoning. The conditions that led to a given warning should be understandable by the maintainer. If the conditions are hidden in complex statistical models, then maintainers cannot review them and will find it difficult to trust the tool.

- Scaling. Industrial sized project contains thousand files and dependencies which can be updated many times a day. Maintenance-oriented tools should not hinder the productivity of maintainers.

- Contextualization. Warnings and messages should always be contextualized with respect to the project at hand and not generic rules.

- Integration. Developers and maintainers are overwhelmed by the amount of existing tools. Yet, their daily use three different kinds of tools. An integrated development environment, a versioning system and a project tracking system to produce, version and manage their software, respectively. Maintenance-oriented tools should fit in the existing ecosystem rather than complexifying the deployment process.

In this chapter, we present `PRECCINT` (PREventing Clones INsertion at Commit Time) and `RESEMBLE` (REcommendation System based on cochangE Mining at Block LEvel) that have the characteristics described by human maintainer.

5.1 PRECINCT: PREventing Clones INsertion at Commit Time

In this section, we present PRECINCT (PREventing Clones INsertion at Commit Time) that focuses on preventing the insertion of clones at commit time, i.e., before they reach the central code repository. PRECINCT is an online clone detection technique that relies on the use of pre-commit hooks capabilities of modern source code version control systems. PRECINCT intercepts this modification and analyses its content to see whether a suspicious clone has been introduced or not. A flag is raised if a code fragment is suspected to be a clone of an existing code segment. In fact, PRECINCT, itself, can be seen as a pre-commit hook that detects clones that might have been inserted in the latest changes with regard to the rest of the source code. This said, only a fraction of the code is analysed, making PRECINCT efficient compared to leading clone detection techniques such as NICAD (Accurate Detection of Near-miss Intentional Clones) [CR11]. Moreover, the detected clones are presented using a classical 'diff' output that developers are familiar with. PRECINCT is also well integrated with the workflow of the developers since it is used in conjunction with a source code version control systems such as Git[1].

In this study, we focus on Type 3 clones as they are more challenging to detect. Since Type 3 clones include Type 1 and 2 clones, then these types could be detected separately by PRECINCT as well.

PRECINCT aims to prevent clone insertion while integrating the clone detection process in a transparent manner in the day-to-day maintenance process. This way, software developers do not have to resort to external tools to remove clones after they are inserted such as the one presented in Section 2.5. Our approach operates at commit time, notifying software developers of possible clones as they commit their code.

We evaluated the effectiveness of PRECINCT using precision and recall on three systems, developed independently and written in both C and Java. The results show that PRECINCT prevents Type 3 clones to reach the final source code repository with an average accuracy of 97.7%.

---

[1]https://git-scm.com/
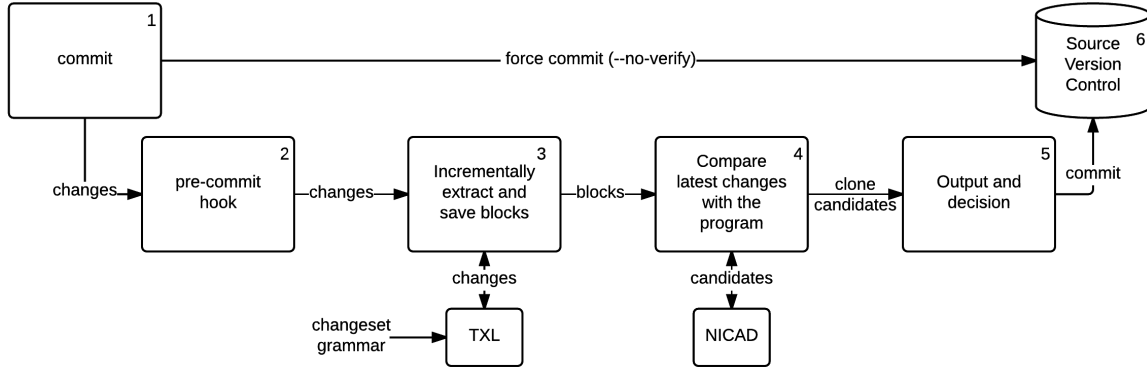
Figure 16: Overview of the PRECINCT Approach.

5.1.1 The PRECINCT Approach

The PRECINCT approach is composed of six steps. The first and last steps are typical steps that a developer would do when committing code. Indeed, the first step is the commit step where developers send their latest changes to the central repository and the last step is the reception of the commit by the central repository. The second step is the pre-commit hook, which kicks in as the first operation when one wants to commit. The pre-commit hook has access to the changes in terms of files that have been modified, more specifically, the lines that have been modified. The modified lines of the files are sent to TXL[Cor06] for block extraction. Then, the blocks are compared to previously extracted blocks in order to identify candidate clones using the comparison engine of NICAD[CR11]. We chose NICAD engine because it has been shown to provide high accuracy [CR11]. The tool is also readily available, easy to use, customizable, and works with TXL. Note, however, that PRECINCT can also work with other engines for comparing code fragments. Finally, the output of NICAD is further refined and presented to the user for decision. These steps are discussed in more detail in the following subsections.

5.1.2 PRECINCT Pre-Commit Hook

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

56

PRECINCT is a set of bash scripts where the entry point of these scripts lies in the pre-commit hooks. Pre-commit hooks are easy to create and implement as depicted in Listing 5.1. This pre-hook is shipped with Git, a popular version control system. Note that even though we use Git as the main version control to present PRECINCT, we believe that the techniques presented in this section are readily applicable to other version control systems. In Listing 5.1, from lines 3 to 11, the script identifies if the commit is the first one in order to select the revision to work against. Then, in Lines 18 and 19, the script checks for trailing whitespace and fails if any are found.

For PRECINCT to work, we just have to add the call to our script suite instead or in addition of the whitespace check.

Listing 5.1: Git Pre-Commit Hook Sample

```
#!/ bin /sh                                              1
                                                         2
if git rev−parse −−verify HEAD > \                       3
 /dev/ null 2>&1                                          4
then                                                     5
 against=HEAD                                            6
else                                                    7
 # Initial commit: diff against                          8
 # an empty tree object                                  9
 against=4b825dc642 ....                                 10
fi                                                      11
                                                        12
# Redirect output to stderr .                            13
exec 1>&2                                               14
                                                        15
# If there are whitespace errors ,                       16
# print the offending file names and fail .              17
exec git diff−index −−check \                            18
        −−cached $against −−                             19
```

5.1.3 Extract and Save Blocks

A block is a set of consecutive lines of code that will be compared to all other blocks in order to identify clones. To achieve this critical part of PRECINCT, we rely on TXL[Cor06], which is a first-order functional programming over linear term rewriting, developed by

57

Cordy et al.[Cor06]. For TXL to work, one has to write a grammar describing the syntax of the source language and the transformations needed. TXL has three main phases: *parse, transform, unparse*. In the parse phase, the grammar controls not only the input but also the output form. Listing 5.2 — extracted from the official documentation[2] — shows a grammar matching an *if-then-else* statement in C with some special keywords: [IN] (indent), [EX] (exdent) and [NL] (newline) that will be used for the output form.

Listing 5.2: Txl Sample Sample

```
define if_statement                          1
  if ( [expr] ) [IN][NL]                     2
    [statement] [EX]                         3
    [opt else_statement]                     4
end define                                   5
                                             6
define else_statement                        7
  else [IN][NL]                              8
    [statement] [EX]                         9
end define                                   10
```

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

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

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

---

[2]http://txl.ca

Data: *Changeset*[] changesets;

Block[] prior_blocks;

Boolean compare_history;

Result: Up to date blocks of the systems

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

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

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

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

            delete *prior_block*;

        end

        write *blocks[j]*;

    end

end

Function *extract_blocks(Changeset cs)*

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

        $cs \leftarrow expand\_left(cs)$;

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

        $cs \leftarrow expand\_right(cs)$;

    end

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

Algorithm 2: Overview of the Extract Blocks Operation

59

Listing 5.3: Changeset c4016c of monit

```
@@ −315,36 +315,6 @@                                    1
int  initprocesstree_sysdep                              2
    ( ProcessTree_T **reference ) {                       3
        mach_port_deallocate(mytask,                      4
            task);                                        5
    }                                                     6
}                                                         7
−  if ( task_for_pid (mytask, pt[i].pid,                  8
−      &task) == KERN_SUCCESS) {                          9
−    mach_msg_type_number_t     count;                   10
−    task_basic_info_data_t     taskinfo;               11
−    thread_array_t             threadtable;            12
−    unsigned int               threadtable_size;       13
−    thread_basic_info_t        threadinfo;             14
```

Algorithm 2 presents an overview of the "extract" and "save" blocks operations of PRECINCT. This algorithm receives as arguments, the changesets, the blocks that have been previously extracted and a boolean named compare_history. Then, from Lines 1 to 9 lie the *for* loop that iterates over the changesets. For each changeset (Line 2), we extract the blocks by calling the *extract_blocks*(*Changeset cs*) function. In this function, we expand our changeset to the left and to the right in order to have a complete block.

As depicted by Listing 5.3, changesets contain only the modified chunk of code and not necessarily complete blocks. Indeed, we have a block from Line 3 to Line 6 and deleted lines from Line 8 to 14. However, in Line 7 we can see the end of a block, but we do not have its beginning. Therefore, we need to expand the changeset to the left in order to have syntactically correct blocks. We do so by checking the block's beginning and ending (using { and }) in C for example. Then, we send these expanded changesets to TXL for block extraction and formalization.

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

We believe that deleted blocks have been deleted for a good reason (bug, default, re-moved features, . . .) and if a newly inserted block matches an old one, it could be worth

knowing in order to improve the quality of the system at hand. This feature is deactivated by default.

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

5.1.4 Compare Extracted Blocks

In order to compare the extracted blocks and detect potential clones, we can only resort to text-based techniques. This is because lexical and syntactic analysis approaches (alternatives to text-based comparisons) would require a complete program to work, a program that compiles. In the relatively wide-range of tools and techniques that exist to detect clones by considering code as text[Joh93; Joh94; MM; Man94; DRD; WM05], we selected NICAD as the main text-based method for comparing clones [CR11] for several reasons. First, NICAD is built on top of TXL, which we also used in the previous step. Second, NICAD is able to detect all Types 1, 2 and 3 software clones.

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

In the *Comparison* phase, extracted blocks are transformed, clustered and compared in order to find potential clones. Using TXL sub-programs, blocks go through a process called pretty-printing where they are stripped of formatting and comments. When code fragments are cloned, some comments, indentation or spacing are changed according to the new context where the new code is used. This pretty-printing process ensures that all code will have the same spacing and formatting, which renders the comparison of code fragments easier. Furthermore, in the pretty-printing process, statements can be broken down into several lines. Table 3 shows how this can improve the accuracy of clone detection with three `for` statements, `for (i=0; i<10; i++)`, `for (i=1; i<10; i++)` and `for (j=2; j<100; j++)`. The pretty-printing allows NICAD to detect Segments 1 and 2 as a clone pair because only the initialization of i changed. This specific example would not have been marked as a clone by other tools we tested such as Duploc[DRD99]. In addition to the pretty-printing, code can be normalized and filtered to detect different classes of clones and match user preferences.

Finally, the extracted, pretty-printed, normalized and filtered blocks are marked as

61

Table 3: Pretty-Printing Example[CHA09]

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

potential clones using a Longest Common Subsequence (LCS) algorithm[HS77]. Then, a percentage of unique statements can be computed and, depending on a given threshold (see Section 5.1.6), the blocks are marked as clones.

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

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

5.1.5 Output and Decision

In this final step, we report the result of the clone detection at commit time with respect to the latest changes made by the developer. The process is straightforward. Every change made by the developer goes through the previous steps and is checked for the introduction of potential clones. For each file that is suspected to contain a clone, one line is printed to the command line with the following options: (I) Inspect, (D) Disregard, (R) Remove from the commit as shown by Figure 17. In comparison to this simple and interactive output, NICAD outputs each and every detail of the detection result such as the total number of potential clones, the total number of lines, the total number of unique line text chars, the total number of unique lines, and so on. We think that so many details might make it hard for developers to react to these results. A problem that was also raised by Johnson et al. [JSMHB13] when examining bug detection tools. Then the potential clones are stored in XML files that can be viewed using an Internet browser or a text editor.

(I) Inspect will cause a diff-like visualization of the suspected clones while (D) disregard

Figure 17: PRECINCT output when replaying commit `710b6b4` of the Monit system used in the case study.

will simply ignore the finding. To integrate PRECINCT in the workflow of the developer we also propose the remove option (R). This option will simply remove the suspected file from the commit that is about to be sent to the central repository. Also, if the user types an option key twice, e.g., II, DD or RR, then the option will be applied to all files. For instance, if the developer types DD at any point, the PRECINCT's results will be disregarded and the commit will be allowed to go through. We believe that this simple mechanism will encourage developers to use PRECINCT like they would use any other feature of Git (or any other version control system).

5.1.6 Case Study

In this section, we show the effectiveness of PRECINCT for detecting clones at commit time in three open source systems[3].

The aim of the case study is to answer the following question: *Can we detect clones at commit time, i.e., before they are inserted in the final code, if so, what would be the accuracy compared to a traditional clone detection tool such as NICAD?*

Target Systems

Table 4 shows the systems used in this study and their characteristics in terms of the number files they contain and the size in KLoC (Kilo Lines of Code). We also include the number of revisions used for each system and the programming language in which the system is written.

Monit[4] is a small open source utility for managing and monitoring Unix systems. Monit

---

[3]The programs used and instructions to reproduce the experiments are made available for download from https://research.mathieu-nayrolles.com/precinct/

[4]https://mmonit.com/monit/

Table 4: List of Target Systems in Terms of Files and Kilo Line of Code (KLOC) at current version and Language

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

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

JHotDraw[5] is a Java GUI framework for technical and structured graphics. It has been developed as a "design exercise". Its design relies heavily on the use of design patterns. JHotDraw is composed of 735 revisions, 1984 files, and the latest revision has 44 KLoC. It is written in Java and it is often used by researchers as a test bench. JHotDraw was also used by NICAD's developers to evaluate their approach.

Dnsjava[6] is a tool for implementing the DNS (Domain Name Service) mechanisms in Java. This tool can be used for queries, zone transfers, and dynamic updates. It is not as large as the other two, but it still makes an interesting case subject because it has been well maintained for the past decade. Also, this tool is used in many other popular tools such as Aspirin, Muffin and Scarab. Dnsjava is composed of 1637 revisions, 233 files, the latest revision contains 47 KLoC.

5.1.7 Process

Figure 18 shows the process we followed to validate the effectiveness of PRECINCT.

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

---

[5]http://www.jhotdraw.org/
[6]http://www.dnsjava.org/
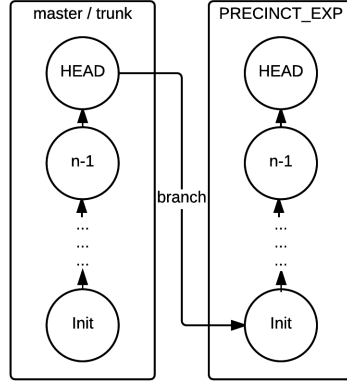
Figure 18: PRECINCT Branching.

To validate the results obtained by PRECINCT, we needed to use a reliable clone detection approach to extract clones from the target systems and use these clones as a baseline for comparison. For this, we turned to NICAD because of its popularity, high accuracy, and availability [CR11], as discussed before. This means, we run NICAD on the revisions of the system to obtain the clones then we used NICAD clones as a baseline for comparing the results obtained by PRECINCT.

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

We show the result of detecting Type 3 clones with a maximum line difference of 30% as discussed in Table 5. As discussed in the introductory section, we chose to report on Type 3 clones because they are more challenging to detect than Type 1 and 2. PRECINCT detects Type 1 and 2 too so does NICAD. For the time being, PRECINCT is not designed to detect Type 4 clones. These clones use different implementations. Detecting Type 4 clones is part of future work.

We assess the performance of PRECINCT in terms of precision (Equation 1) and recall (Equation 2). Both precision and recall are computed by considering NICAD's results as a baseline. We also compute F_1-measure (Equation 3), i.e., the weighted average of precision and recall, to measure the accuracy of PRECINCT.

$$precision = \frac{|\{NICAD_{detection}\} \cap \{PRECINCT_{detection}\}|}{|\{PRECINCT_{detection}\}|} \qquad (8)$$

65
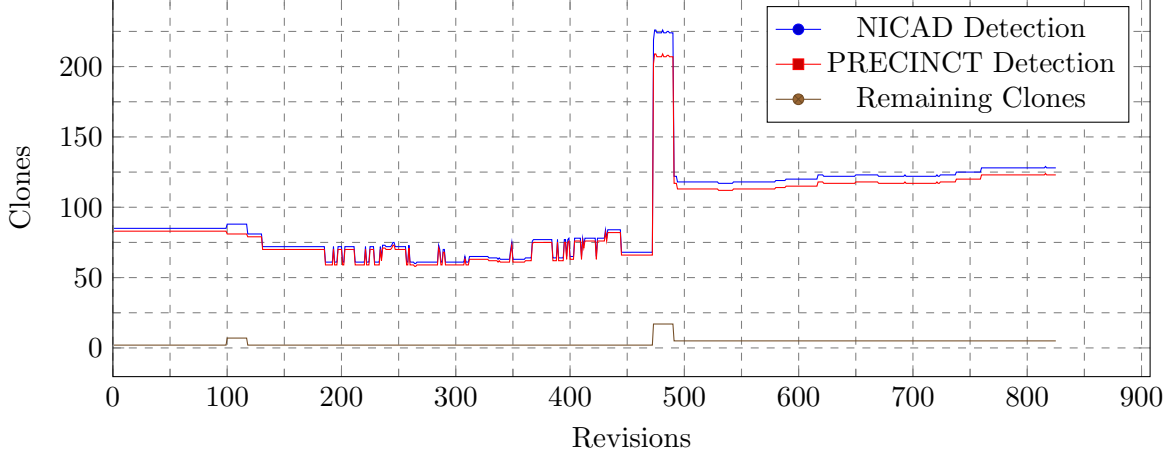
Figure 19: Monit clone detection over revisions
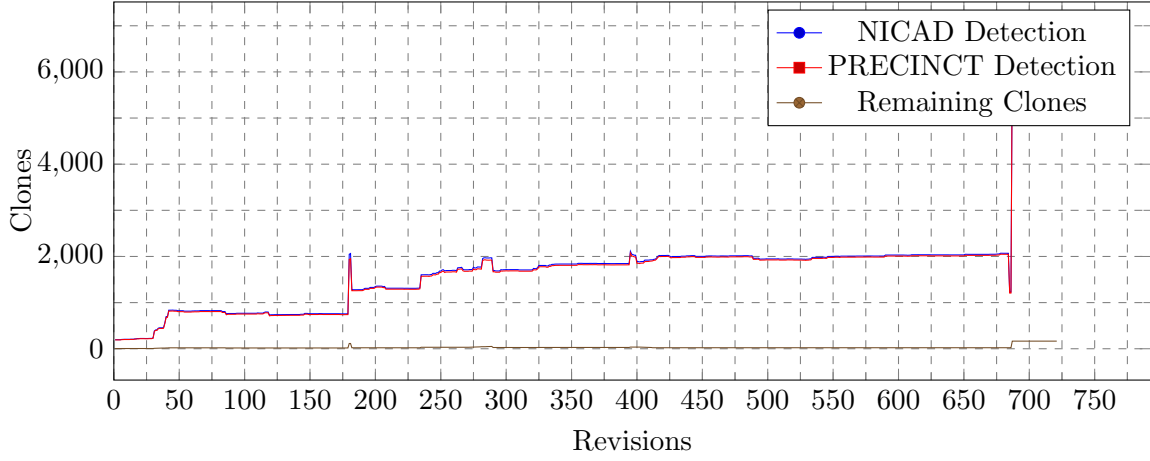
Figure 20: JHotDraw clone detection over revisions

$$recall = \frac{|\{NICAD_{detection}\} \cap \{PRECINCT_{detection}\}|}{|\{NICAD_{detection}\}|} \qquad (9)$$

$$F_1 - measure = 2 * \frac{precision * recall}{precision + recall} \qquad (10)$$

5.1.8 Results

Figures 19, 20, 21 show the results of our study in terms of clone pairs that are detected per revision for our three subject systems: Monit, JHotDraw and Dnsjava. We used as baseline for comparison the clone pairs detected by NICAD. The blue line shows the clone detection performed by NICAD. The red line shows the clone pairs detected by PRECINCT.

66
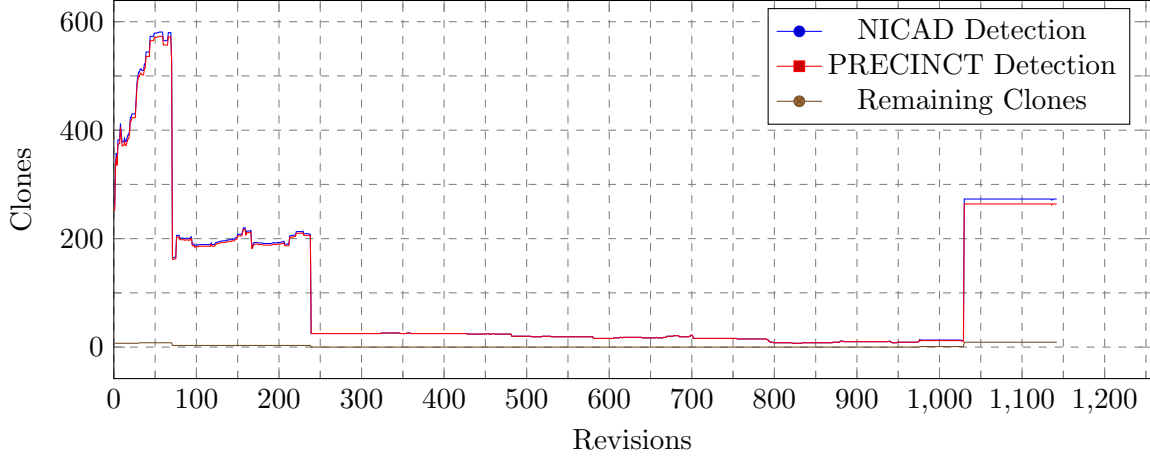
Figure 21: Dnsjava clone detection over revisions

Table 5: Overview of PRECINCT's results in terms of precision, recall, F_1-measure, execution time and output reduction.

	NICAD	PREC.	Prec.	Recall	F1	NICAD' Time	PREC.'s Time	Output Reduc.
Monit	128	123	96.1%	100%	98%	2.2s	0.9s	88.3%
JHotDraw	6599	6490	98.3%	100%	99.1%	5.1s	1.7s	70.1%
DnsJava	273	226	82.8%	100%	90.6%	1.8s	1.1s	88.6%
Total	7000	6839	97.7%	100%	98.8%	3s	1.2s	83.4%

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

Table 5 summarizes PRECINCT's results in terms of precision, recall, F_1-measure, execution time and output reduction. The first version of Monit contains 85 clone pairs and this number stays stable until Revision 100. From Revision 100 to 472 the detected clone pairs vary between 68 and 88 before reaching 219 at Revision 473. The number of clone pairs goes down to 122 at Revision 491 and decreases to 128 in the last revision. PRECINCT was able to detect 96.1% (123/128) of the clone pairs that are detected by NICAD with a 100% recall. It took in average around 1 second for PRECINCT to execute on a Debian 8 system with Intel(R) Core(TM) i5-2400 CPU @ 3.10GHz, 8Gb of DDR3 memory. It is also worth mentioning that the computer we used is equipped with SSD (Solid State Drive). This impacts the running time as clone detection is a file intensive operation. Finally, the PRECINCT was able to output 88.3% less lines than NICAD.

67

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

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

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

The difference in execution time between NICAD and PRECINCT stems from the fact that, unlike PRICINCT, NICAD is not an incremental approach. For each revision, NICAD has to extract all the code blocks and then compares all the pairs with each other. On the other hand, PRECINCT only extracts blocks when they are modified and only compares what has been modified with the rest of the program.

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

5.1.9 Threats to Validity

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

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

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

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

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

5.1.10 Conclusion

PRECINCT (PREventing Clones INsertion at Commit Time) is an incremental approach for preventing clone insertion at commit time that combines efficient block extraction and clone detection and integrate itself seamlessly in the day-to-day workflow of developers. PRECINCT takes advantage of TXL and NICAD to create a clone detection tool approach that runs automatically before each commit in 1.2 second with a 97.7% precision and a 100% recall (when using NICAD results as a baseline).

Our approach also assesses two major factors that contribute to the slow adoption of clone detection tools: large number of data output by clone detection methods, and smooth integration with the task flow of the developers. PRECINCT is able to reduce the number of lines output by a classical clone detection tool such as NICAD by 83.4% while keeping all the necessary information that allow developers to decide whether the detect clone is in fact a clone. Also, our approach is seamlessly integrated with the developers' workflow by means of pre-commit hooks, which are part any version control systems.

To build on this work, we need to experiment with additional (and larger) systems with the dual aim to (a) improve and fine-tune the approach, and (b) assess the scalability of our approach when applied to even larger (and proprietary) systems. Also, we want to improve PRECINCT to support Type 4 clones.

Using the block extracted by `PRECINCT`, we can recommend actions to the developer. To do so, we plan to build `RESEMBLE`. `RESEMBLE` is presented in the next section.

5.2 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 the form of another pre-commit hook. `RESEMBLE` will leverage the branches' history and the blocks extracted by `PRECINCT` to recommend blocks that should be modified. More specifically, `RESEMBLE` will mine the blocks extracted by `PRECINCT` in order to establish sequences of changes. On new changes, the previously established sequence can be compared to the new ones and recommendation made about which blocks you have been modified.

In the data mining field, association rule mining (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 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 are 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 will use a sequential pattern/rule miner that has the advantage to be freely available on the web[7]. Although it has not been optimized for pure sequences its performances are more than satisfactory.

We did not started the experiments for `RESEMBLE` yet.

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 at commit-time, using the local history and ressources, 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.

In summary, using `PRECINCT` and `RESSEMBLE` we are able to provide contextualized and accurate information in a non-intrusive manner.

---

[7]http://www.philippe-fournier-viger.com/spmf/

Chapter 6

Preventing Bug Insertion Using Clone Detection

`BIANCA` (Bug Insertion ANticipation by Clone Analysis at merge 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.

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.

`BIANCA` is different than the approaches presented in the previous sections because it 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). Also, `BIANCA` is an offline approach that is triggered by a merge request. When a maitainers estimate that their work are ready to be integrated with the main branch, they open a merge request[1]. Merging a task branch is not an instanteous process as the code need to pass code review. `BIANCA` leverages this *down* time to perform a complete history check on all projects contained in `BUMPER`.

Figure 22 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,

---

[1]Also known as pull request
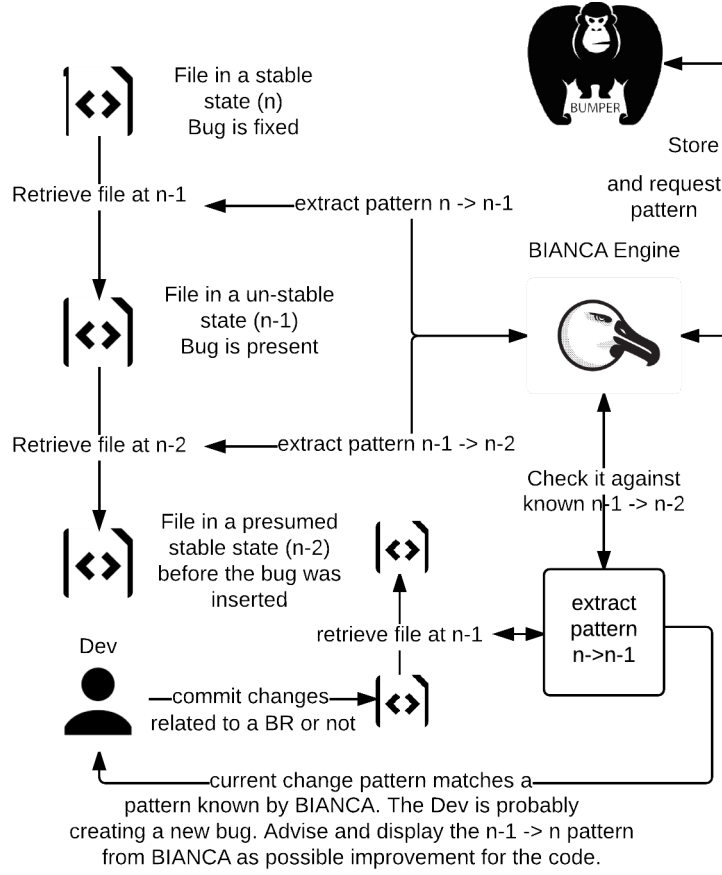
Figure 22: The BIANCA Approach

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 the step to reproduce it.

To extract the change patterns and compare them, we used the same technique as the one presented in section 5.1.3. The third and the fourth normalizations are removing all

73

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].

6.1   Early experiments

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

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 6: 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 **??**. 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 23.

74
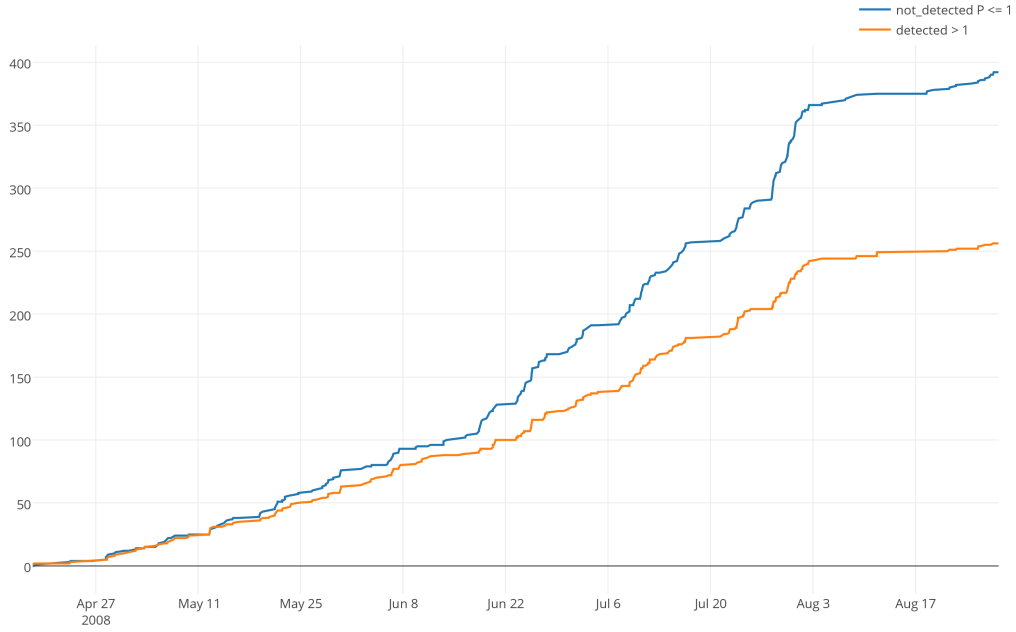
Figure 23: `BIANCA` warnings from April to August 2008 using the first normalization.

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%) commit we analyze. However, the false positive rate increases to 21%. Figure 24 shows the results.

`BIANCA` experiments are still in their early stage and we are still trying to improve our normalizations in order to reduce the false positive rate.
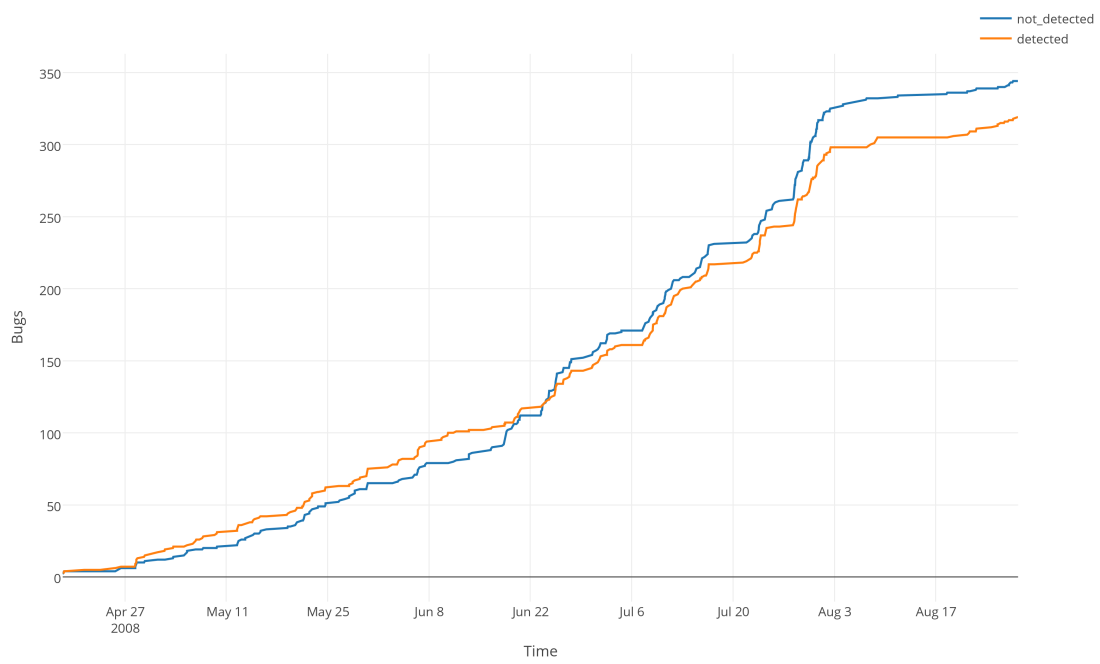
Figure 24: `BIANCA` warnings from April to August 2008 using the second normalization.

Chapter 7

Remaining Work to Complete the Thesis

In this chapter, we summarize the state of the research and the work that needs to be completed in order to finalize the thesis. We proceed according to the contributions listed on Section 1.1.

7.1 An aggregate bug repository for developers and Researchers

We introduced `BUMPER` (BUg Metarepository for dEvelopers and Researchers), a web-based infrastructure that can be used by software developers and researchers to access data from diverse repositories using natural language queries in a transparent manner, regardless of where the data was originally created and hosted. `BUMPER` have been showcased in the following publications:

- Nayrolles, M. & Hamou-Lhadj, W. BUMPER: A Tool to Cope with Natural Language Search of Millions Bugs and Fixes. In Proceeding of the International Conference on Software Analysis, Evolution, and Reengineering (SANER'16) - Tool Track, pages 649-652, 2016.

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

We consider this contribution to be 100% complete.

7.2 A bug reproduction technique based on a combination of crash traces and model checking.

In this work, we proposed 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. `JCHARMING` have been showcased in the following publications:

- 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. 2016. (Accepted).

- Nayrolles, M. , Hamou-Lhadj, W., Tahar, S. Larsson, A. JCHARMING : A Bug Reproduction Approach Using Crash Traces and Directed Model Checking. In Proceeding of the International Conference on Software Analysis, Evolution, and Reengineering (SANER'15), pages 101-110, 2015. (Best Paper Award).

We consider this contribution to be 100% complete.

7.3 An incremental approach for preventing bug and clone insertion at commit time

We presented `PRECCINT` (PREventing Clones INsertion at Commit Time), `RESEMBLE` (REcommendation System based on cochangE Mining at Block LEvel) and `BIANCA` (Bug Insertion ANticipation by Clone Analysis at merge time) in chapters 5 and 6.

The efficiency of `PRECCINT` have been accessed. Early experiments have been conducted for `BIANCA`. However, we need to conduct additional experiments to measure the effiency `RESEMBLE` and `BIANCA`.

We consider this contribution to be 40% complete.

7.4 A new taxonomy of bugs based on the location of the correction — an empirical Study

In order to classify the research on the different fields related to software maintenance, 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.

Our aim is not to improve testing as it is the case in the work of Eldh [Eld01] and Hamill et al.[HGP14]. Our objective is to propose a classification that can allow researchers in the filed of mining bug 9 repositiories to use the taxonomy as a new criterion in triaging, prediction, and reproduction of bugs. By analogy, we can look at the proposed bug taxonomy in a similar way as the clone taxonomy presented by Kapser and Godfrey [Cor]. The authors proposed seven types of source code clones and then conducted a case study, using their classification, on the file system module of the Linux operating system. This clone taxonomy continues to be used by researchers to build better approaches for detecting a given clone type and being able to effectively compare approaches with each other.

In this section, 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 25. 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 26).

Figure 25: Class diagram showing the relationship between bugs and fixed
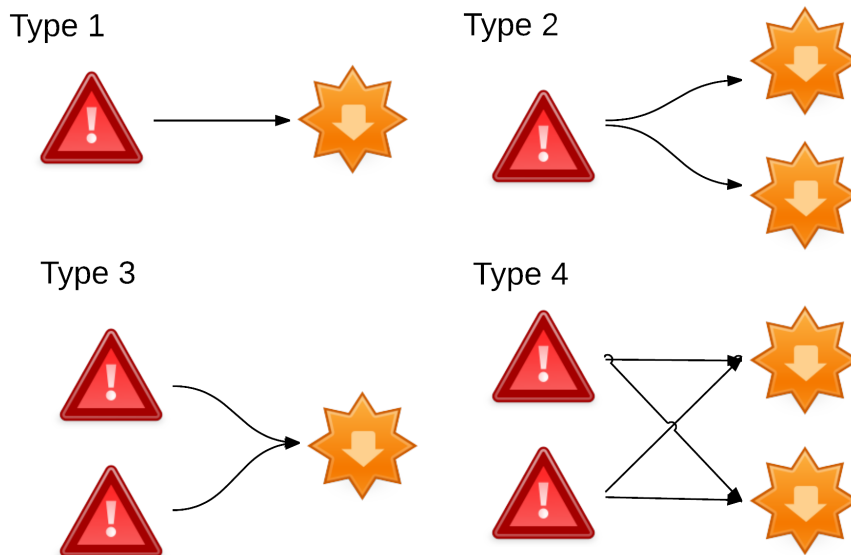

Figure 26: Proposed Taxonomy of Bugs

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 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 section. 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?

7.4.1 Study Setup

Figure 27 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).

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).

We consider this contribution to be 60% complete.
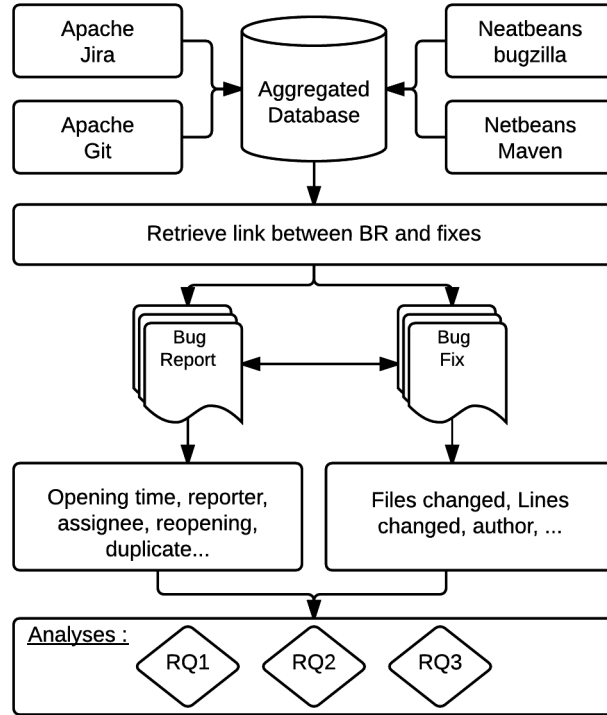
Figure 27: Data collection and analysis process of the study

7.5 Publication Plan

This section presents our past and future publications. Publications 1 to 5 have already been accepted in various international conferences and journals.

- **Publication 1**. A Bug Reproduction Approach Using Crash Traces and Directed Model Checking. In Proceeding of the International Conference on Software Analysis, Evolution, and reengineering (SANER'15), pages 101-110, 2015. (Best Paper Award).

- **Publication 2**. An Empirical Study on the Handling of Crash Reports in a Large Software Company: An Experience Report. In Proceeding of the IEEE International Conference on Software Maintenance and Evolution (ICSME'15), pages 342-351, 2015. (Third author)

- **Publication 3**. BUMPER: Bug Metarepository Search Engine for Developers and Researchers. Consortium for Software Engineering Research Fall (CSER'15), 2015.

- **Publication 4**. BUMPER: A Tool to Cope with Natural Language Search of Millions Bugs and Fixes. In Proceeding of the International Conference on Software Analysis, Evolution, and Reengineering (SANER'16) - Tool Track, pages 649-652, 2016.

- **Publication 5**. A Bug Reproduction Approach Based on Directed Model Checking and Crash Traces. Journal of Software: Evolution and Process. Wiley. 2016. (Accepted).

- **Publication 6**. A New Taxonomy of Bugs Based on the Location of the Correction: An Empirical Study will be submitted to IEEE Transaction on Software Engineering.

- **Publication 7**. JAPPA: An Automatic Patch Generation Approach Using Model Checking and Changesets History will be submitted to the Empirical Software Engineering Journal (EMSE) special issue on Automatic Software Repair.

- **Publication 8**. PRECINCT: An Incremental Approach for Preventing Clone Insertion at Commit Time will be submitted to the Journal of Systems and Software.

- **Publication 9**. BIANCA: Bug Insertion ANticipation by Clone Analysis at commit-time will be submitted to IEEE Software special issue on Theme Issue on Reliability Engineering for Software.

- **Publication 10**. RESEMBLE: REcommendation System based on cochangE Mining at Block LEvel will be submitted to SANER'17

- **Publication 11**. Software Maintenance at Commit-Time will be submitted to IEEE Transaction on Software Engineering.

- **Thesis**. In parallel to publications 9 and 10, I plan to write my Ph.D thesis.

Table 7 presents an overview of the publications planning. For each contribution, the title, the target conference or journal and the time at which we plan to work on it is given. The red line represents the current time.

Table 7: Publications Plan

Title	Target	J	F	M	A	M	J	J	A	O	N	D
2014												
Courses Requirements	n.a	█	█	█	█	█	█	█				
Literature Review	n.a			█	█	█	█	█				
A Bug Reproduction Approach Using Crash Traces and Directed Model Checking	SANER							█	█	█	█	█
2015												
An Empirical Study on the Handling of Crash Reports in a Large Software Company: An Experience Report.	ICSME	█	█	█								
BUMPER: Bug Metarepository Search Engine for Developers and Researchers.	CSER				█	█	█	█	█	█	█	█
BUMPER: A Tool to Cope with Natural Language Search of Millions Bugs and Fixes.	SANER				█	█	█	█	█	█	█	█
A Bug Reproduction Approach Based on Directed Model Checking and Crash Traces.	JSEP				█	█	█	█	█	█	█	█
2016												
A New Taxonomy of Bugs Based on the Location of the Correction: An Empirical Study	TSE	█	█	█	█	█	█					
Preparing the Ph.D research proposal	n.a	█	█	█	█	█	█					
JAPPA: An Automatic Patch Generation Approach Using Model Checking and Changesets History	EMSE					█	█	█	█			
PRECINCT: An Incremental Approach for Preventing Clone Insertion at Commit Time	JSS					█	█	█	█			
Bug Insertion ANticipation by Clone Analysis at commit-time	IEEE SW									█	█	█
2017												
RESEMBLE: REcommendation System based on cochangE Mining at Block LEvel	SANER	█	█	█	█							
Software Maintenance at Commit-Time	IEEE SW	█	█	█	█							
Write the Ph.D Thesis	n.a				█	█	█	█				

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.

[AHM⁺08] N. Ayewah, D. Hovemeyer, J.D. Morgenthaler, J. Penix, and W. Pugh. Using Static Analysis to Find Bugs. *IEEE Software*, 25(5):22–29, 2008.

[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.

[ALRL04] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, jan 2004.

[AP08] Nathaniel Ayewah and William Pugh. A report on a survey and study of static analysis users. In *Proceedings of the 2008 workshop on Defects in large software systems - DEFECTS '08*, page 1, New York, New York, USA, jul 2008. ACM Press.

[AP10] Nathaniel Ayewah and William Pugh. The Google FindBugs fixit. In *Proceedings of the 19th international symposium on Software testing and analysis - ISSTA '10*, page 241, New York, New York, USA, jul 2010. ACM Press.

[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.

[APM⁺07] Nathaniel Ayewah, William Pugh, J. David Morgenthaler, John Penix, and YuQian Zhou. Evaluating static analysis defect warnings on production software. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering - PASTE '07*, pages 1–8, New York, New York, USA, jun 2007. ACM Press.

[ASWF14] Satoshi Arai, Kazunori Sakamoto, Hironori Washizaki, and Yoshiaki Fukazawa. A gamified tool for motivating developers to remove warnings of bug pattern tools. In *Proceedings - 2014 6th International Workshop on Empirical Software Engineering in Practice, IWESEP 2014*, pages 37–42, 2014.

[Bak] B.S. Baker. On finding duplication and near-duplication in large software systems. In *Proceedings of 2nd Working Conference on Reverse Engineering*, pages 86–95. IEEE Comput. Soc. Press.

[Bak92] Brenda S. Baker. A program for identifying duplicated code. *Computing Science and Statistics*, 1992.

[BBKE13] Brian Burg, Richard Bailey, Andrew J. Ko, and Michael D. Ernst. Interactive record/replay for web application debugging. In *Proceedings of the 26th annual ACM symposium on User Interface Software and Technology*, pages 473–484, New York, New York, USA, oct 2013. ACM Press.

[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.

[BG02]   Brenda S. Baker and Raffaele Giancarlo.   Sparse Dynamic Programming
           for Longest Common Subsequence from Fragments. *Journal of Algorithms*,
           42(2):231–254, feb 2002.

[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 Foun-
           dations of Software Engineering*, page 308, New York, New York, USA, 2008.
           ACM Press.

[BKA+07]   Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore
           Merlo. Comparison and Evaluation of Clone Detection Tools. *IEEE Transac-
           tions on Software Engineering*, 33(9):577–591, sep 2007.

[BKB+06]   Laura Beckwith, Cory Kissinger, Margaret Burnett, Susan Wiedenbeck,
           Joseph Lawrance, Alan Blackwell, and Curtis Cook. Tinkering and gender
           in end-user programmers' debugging. *Proceedings of the SIGCHI conference
           on Human Factors in computing systems - CHI '06*, page 231, 2006.

[BMD+]   M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis. Partial
           redesign of Java software systems based on clone analysis. In *Sixth Working
           Conference on Reverse Engineering (Cat. No.PR00303)*, pages 326–336. IEEE
           Comput. Soc.

[BMD+99]   M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis. Mea-
           suring clone based reengineering opportunities. In *Proceedings Sixth Interna-
           tional Software Metrics Symposium (Cat. No.PR00403)*, pages 292–303. IEEE
           Comput. Soc, 1999.

[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.

[Bug08]   Bugzilla. Life Cycle of a Bug, 2008.

[Bur06]   Ilene Burnstein. *Practical Software Testing: A Process-Oriented Approach.*
           Springer Science & Business Media, 2006.

[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.

[BYM+98] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone Detection Using Abstract Syntax Trees. In *Proceedings of the IEEE International Conference on Software Maintenance.*, pages 368–377. IEEE Computer Society, mar 1998.

[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.

[CHA09] CHANCHAL K. ROY. *Detection and Analysis of Near-Miss Software Clones.* PhD thesis, Queen's University, 2009.

[Che13a] Ning Chen. *Star: stack trace based automatic crash reproduction.* PhD thesis, The Hong Kong University of Science and Technology, 2013.

[Che13b] Ning Chen. *Star: stack trace based automatic crash reproduction.* PhD thesis, Honk 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.

[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.

[Cor06] 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.

[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.

[DB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[DCMS] Thomas R. Dean, James R. Cordy, Andrew J. Malton, and Kevin A. Schneider. Agile Parsing in TXL. In *Proceedings of IEEE International Conference on Automated Software Engineering*, volume 10, pages 311–336. Kluwer Academic Publishers.

[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.

[DER07] Ekwa Duala-Ekoko and Martin P. Robillard. Tracking Code Clones in Evolving Software. In *29th International Conference on Software Engineering*, pages 158–167. IEEE, may 2007.

[DER10] Ekwa Duala-Ekoko and Martin P. Robillard. Clone region descriptors. *ACM Transactions on Software Engineering and Methodology*, 20(1):1–31, jun 2010.

[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.

[DRD] Stephane Ducasse, Matthias Rieger, and Serge Demeyer. A Language Independent Approach for Detecting Duplicated Code.

[DRD99] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM'99). 'Software Maintenance for Business Change' (Cat. No.99CB36360)*, pages 109–118. IEEE, 1999.

[Eld01] Sigrid Eldh. *On Test Design*. PhD thesis, Mälardalen, 2001.

[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.

[EPHJ07] S. Eldh, S. Punnekkat, H. Hansson, and P. Jönsson. Component testing is not enough-a study of software faults in telecom middleware. *Testing of Software and Communicating Systems*, 4581:74–89, 2007.

[FFK08] Raimar Falke, Pierre Frenzel, and Rainer Koschke. Empirical evaluation of clone detection using syntax suffix trees. *Empirical Software Engineering*, 13(6):601–643, jul 2008.

[Fin15] Findbugs. FindBugs Bug Descriptions, 2015.

[FM15] Sylvie L. Foss and Gail C. Murphy. Do developers respond to code stability warnings? pages 162–170, nov 2015.

[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.

[GJS08] Mark Gabel, Lingxiao Jiang, and Zhendong Su. Scalable detection of semantic clones. In *Proceedings of the 13th international conference on Software engineering - ICSE '08*, page 321, New York, New York, USA, 2008. ACM Press.

[GM04] Victor M González and Gloria Mark. Constant , Constant , Multi-tasking Craziness : Managing Multiple Working Spheres. *CHI '04 Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 6(1):113–120, 2004.

[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.

[HGP14] Maggie Hamill and Katerina Goseva-Popstojanova. Exploring fault types, detection activities, and failure severity in an evolving safety-critical software system. *Software Quality Journal*, 23(2):229–265, apr 2014.

[HGWB11] Steffen Herbold, Jens Grabowski, Stephan Waack, and Uwe Bünting. Improved Bug Reporting and Reproduction through Non-intrusive GUI Usage Monitoring and Automated Replaying. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 232–241. IEEE, mar 2011.

[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.

[HKKI04] Yoshiki Higo, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Refactoring support based on code clone analysis. In *Product Focused Software Process Improvement*, pages 220–233. Springer, 2004.

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

[Hov07] David Hovemeyer. FindBugs, 2007.

[HP04a] David Hovemeyer and William Pugh. Finding bugs is easy. *ACM SIGPLAN Notices*, 39(12):92, dec 2004.

[HP04b] David Hovemeyer and William Pugh. Finding bugs is easy. *ACM SIGPLAN Notices*, 39(12):92, 2004.

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

[HS77] James W. Hunt and Thomas G. Szymanski. A fast algorithm for computing longest common subsequences. *Communications of the ACM*, 20(5):350–353, may 1977.

[JDHW09] Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. Do code clones matter? In *2009 IEEE 31st International Conference on Software Engineering*, pages 485–495. IEEE, may 2009.

[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.

[JMSG07] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones. In *29th International Conference on Software Engineering*, pages 96–105. IEEE, may 2007.

[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.

[Joh93] J Howard Johnson. Identifying redundancy in source code using fingerprints. In *CASCON '93 Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research: software engineering - Volume 1*, pages 171–183. IBM Press, oct 1993.

[Joh94] J. Howard Johnson. Visualizing textual redundancy in legacy source. In *Proceedings of the 1994 conference of the Centre for Advanced Studies on Collaborative research*, page 32. IBM Press, oct 1994.

[JSMHB13] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *35th International Conference on Software Engineering (ICSE)*, pages 672–681. IEEE Press, may 2013.

[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.

[KE07a] Sunghun Kim and Michael D. Ernst. Prioritizing warning categories by analyzing software history. In *Proceedings - ICSE 2007 Workshops: Fourth International Workshop on Mining Software Repositories, MSR 2007*, 2007.

[KE07b] Sunghun Kim and Michael D Ernst. Which warnings should I fix first? *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering ESECFSE 07*, page 45, 2007.

[KFF06] Rainer Koschke, Raimar Falke, and Pierre Frenzel. Clone Detection Using Abstract Syntax Suffix Trees. In *2006 13th Working Conference on Reverse Engineering*, pages 253–262. IEEE, oct 2006.

[KG] C. Kapser and M.W. Godfrey. Aiding comprehension of cloning through categorization. In *Proceedings. 7th International Workshop on Principles of Software Evolution, 2004.*, pages 85–94. IEEE.

[KG06] Cory Kapser and Michael Godfrey. Cloning Considered Harmful Considered Harmful. In *2006 13th Working Conference on Reverse Engineering*, pages 19–28. IEEE, oct 2006.

[KH] R. Komondoor and S. Horwitz. Effective, automatic procedure extraction. In *MHS2003. Proceedings of 2003 International Symposium on Micromechatronics and Human Science (IEEE Cat. No.03TH8717)*, pages 33–42. IEEE Comput. Soc.

[KH00] Raghavan Komondoor and Susan Horwitz. Semantics-preserving procedure extraction. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '00*, pages 155–169, New York, New York, USA, jan 2000. ACM Press.

[KKI02a] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, jul 2002.

[KKI02b] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, jul 2002.

[KLHK13] Jinhan Kim, Sanghoon Lee, Seung-Won Hwang, and Sunghun Kim. Enriching Documents with Examples. *ACM Transactions on Information Systems*, 31(1):1–27, jan 2013.

[KMCA06] Andrew J. Ko, Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on Software Engineering*, 32(12):971–987, 2006.

[Kon] K. Kontogiannis. Evaluation experiments on the detection of programming patterns using software metrics. In *Proceedings of the Fourth Working Conference on Reverse Engineering*, pages 44–54. IEEE Comput. Soc.

[Kop06] Timo Koponen. Life cycle of defects in open source software projects. In *Open Source Systems*, pages 195–200. Springer, 2006.

[KPW06] Sunghun Kim, Kai Pan, and E. E. James Whitehead. Memories of bug fixes. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering - SIGSOFT '06/FSE-14*, page 35, New York, New York, USA, nov 2006. ACM Press.

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

[Kri01] Jens Krinke. Identifying Similar Code with Program Dependence Graphs. In *Eighth IEEE Working Conference on Reverse Engineering, 2001*, pages 301–309. IEEE Computer Society, oct 2001.

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

[KSN05] Miryung Kim, Vibha Sazawal, and David Notkin. An empirical study of code clone genealogies. *ACM SIGSOFT Software Engineering Notes*, 30(5):187, sep 2005.

[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.

[KZWZ07]   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.

[Lay09]   Lucas Michael Layman. *Information needs of developers for program comprehension during software maintenance tasks.* PhD thesis, 2009.

[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.

[LHMI07]   Simone Livieri, Yoshiki Higo, Makoto Matushita, and Katsuro Inoue. Very-Large Scale Code Clone Analysis and Visualization of Open Source Programs Using Distributed CCFinder: D-CCFinder. In *29th International Conference on Software Engineering*, pages 106–115. IEEE, may 2007.

[LLMZ06]   Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering*, 32(3):176–192, mar 2006.

[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. In *International Conference on Software Engineering (ICSE) (2013)*, pages 372–381. IEEE Press, may 2013.

[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.

[LPM+]   B. Lague, D. Proulx, J. Mayrand, E.M. Merlo, and J. Hudepohl. Assessing the benefits of incorporating function clone detection in a development process.

In *Proceedings International Conference on Software Maintenance*, pages 314–321. IEEE Comput. Soc.

[LvdH11] Nicolas Lopez and André van der Hoek. The code orb. In *Proceeding of the 33rd international conference on Software engineering - ICSE '11*, page 824, New York, New York, USA, may 2011. ACM Press.

[LWA07] Lucas Layman, Laurie Williams, and Robert St. Amant. Toward Reducing Fault Fix Time: Understanding Developer Behavior for the Design of Automated Fault Detection Tools. In *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*, pages 176–185. IEEE, sep 2007.

[Man94] Udi Manber. Finding similar files in a large file system. In *Usenix Winter*, pages 1–10. USENIX Association, jan 1994.

[Mar09] Martin Fowler. FeatureBranch, 2009.

[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.

[MHJM12] Emerson Murphy-Hill, Rahul Jiresal, and Gail C. Murphy. Improving software developers' fluency by recommending development environment commands. *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering - FSE '12*, page 1, 2012.

[MLM96] Jean Mayrand, Claude Leblanc, and Ettore M. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proceedings of International Conference on Software Maintenance ICSM-96*, pages 244–253. IEEE, 1996.

[MM] A. Marcus and J.I. Maletic. Identification of high-level concept clones in source code. In *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*, pages 107–114. IEEE Comput. Soc.

[MPNC12] Naouel Moha, Francis Palma, Mathieu Nayrolles, and Benjamin Joyen Conseil. Specification and Detection of SOA Antipatterns. In *International Conference on Service Oriented Computing*, pages 1–16. Springer, 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.

[NAW⁺08] Syeda Nessa, Muhammad Abedin, W Eric Wong, Latifur Khan, and Yu Qi. Software Fault Localization Using N -gram Analysis. In *WASA*, pages 548–559, 2008.

[Nay13] Mathieu; Nayrolles. *Improving SOA Antipattern Detection in Service Based Systems by Mining Execution Traces*. PhD thesis, 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.

[NBMV15] Mathieu Nayrolles, Eric Beaudry, Naouel Moha, and Petko Valtchev. Towards Quality-Driven SOA Systems Refactoring through Planning. In *6th International MCETECH Conference*, 2015.

[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.

[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.

[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.

[Nor13]   Donald A. Norman. *The Design of Everyday Things: Revised and Expanded Edition*. Basic Books, 2013.

[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.

[NPK13]   Jaechang Nam, Sinno Jialin Pan, and Sunghun Kim. Transfer defect learning. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 382–391. Ieee, may 2013.

[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.

[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.

[PMDL99] J.-F. Patenaude, E. Merlo, M. Dagenais, and B. Lague. Extending software quality assessment techniques to Java systems. In *Proceedings Seventh International Workshop on Program Comprehension*, pages 49–56. IEEE Comput. Soc, 1999.

[PNM13] Francis Palma, Mathieu Nayrolles, and Naouel Moha. SOA Antipatterns : An Approach for their Specification and Detection. *International Journal of Cooperative Information Systems*, 22(04):1–40, 2013.

[Pra01] Michael J. Pratt. Introduction to ISO 10303the STEP Standard for Product Data Exchange. *Journal of Computing and Information Science in Engineering*, 1(1):102, mar 2001.

[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.

[RAF04] N. Rutar, C.B. Almazan, and J.S. Foster. A Comparison of Bug Finding Tools for Java. In *15th International Symposium on Software Reliability Engineering*, pages 245–256. IEEE, 2004.

[RAN07]   Per Runeson, Magnus Alexandersson, and Oskar Nyholm. Detection of Dupli-
cate 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.

[RGK90]   Jane Radatz, Anne Geraci, and Freny Katki. IEEE standard glossary of soft-
ware engineering terminology. *IEEE Std*, 610121990(121990):3, 1990.

[RLB06]   T. J. Robertson, Joseph Lawrance, and Margaret Burnett. Impact of high-
intensity negotiated-style interruptions on end-user debugging.   *Journal of
Visual Languages and Computing*, 17(2):187–202, 2006.

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

[RNB15]   Tobias Roehm, Stefan Nosovic, and Bernd Bruegge. Automated extraction of
failure reproduction steps from user interaction traces. In *2015 IEEE 22nd
International Conference on Software Analysis, Evolution, and Reengineering
(SANER)*, pages 121–130. IEEE, mar 2015.

[RPB+04]   T J Robertson, Shrinu Prabhakararao, Margaret Burnett, Curtis Cook,
Joseph R Ruthruff, Laura Beckwith, and Amit Phalgune. Impact of inter-
ruption style on end-user debugging. *Proc. CHI*, 6(1):287–294, 2004.

[RYR13]   Mohammad Masudur Rahman, Shamima Yeasmin, and Chanchal K. Roy. An
IDE-based context-aware meta search engine. In *2013 20th Working Confer-
ence 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, IEEE*, 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.

[SFZ11]   Haihao Shen, Jianhong Fang, and Jianjun Zhao. EFindBugs: Effective Error Ranking for FindBugs. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, pages 299–308. IEEE, mar 2011.

[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.

[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.

[TBG]   M. Toomim, A. Begel, and S.L. Graham. Managing Duplicated Code with Linked Editing. In *2004 IEEE Symposium on Visual Languages - Human Centric Computing*, pages 173–180. IEEE.

[TG06]   Robert Tairas and Jeff Gray. Phoenix-based clone detection using suffix trees. In *Proceedings of the 44th annual southeast regional conference on - ACM-SE 44*, page 679, New York, New York, USA, mar 2006. ACM Press.

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

[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.

[VHD07] Mathieu Verbaere, Elnar Hajiyev, and Oege De Moor. Improve software quality with SemmleCode. In *Companion to the 22nd ACM SIGPLAN conference on Object oriented programming systems and applications companion - OOPSLA '07*, page 880, New York, New York, USA, oct 2007. ACM Press.

[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.

[WAC12] James A. Whittaker, Jason Arbon, and Jeff Carollo. *How Google Tests Software.* Addison-Wesley, 2012.

[Wel13] Brian Wellington. dnsjava, 2013.

[WM05] R. Wettel and R. Marinescu. Archeology of code duplication: recovering duplication chains from small duplication fragments. In *Seventh International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC'05)*, page 8 pp. IEEE, 2005.

[WSWF] V. Wahler, D. Seipel, J. Wolff, and G. Fischer. Clone detection in source code by frequent itemset techniques. In *Source Code Analysis and Manipulation, Fourth IEEE International Workshop on*, pages 128–135. IEEE Comput. Soc.

[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.

[WZZ07] Cathrin Weiß, Thomas Zimmermann, and Andreas Zeller. How Long will it Take to Fix This Bug ? In *Fourth International Workshop on Mining Software Repositories (MSR'07)*, number 2, page 1, 2007.

[Xst11] Xstream. Xstream, 2011.

[ZC10] Cristian Zamfir and George Candea. Execution Synthesis: A Technique for Automated Software Debugging. In *Proceedings of the 5th European Conference on Computer Systems*, pages 321–334, 2010.

[Zel97] Andreas Zeller. *Configuration management with version sets: A unified software versioning model and its applications.* 1997.

[Zel13] Andreas Zeller. Where Should We Fix This Bug? A Two-Phase Recommendation Model. *IEEE Transactions on Software Engineering*, 39(11):1597–1610, nov 2013.

[ZGV13] Hongyu Zhang, Liang Gong, and Steve Versteeg. Predicting bug-fixing time: an empirical study of commercial software projects. In *International Conference on Software Engineering*, pages 1042–1051. IEEE Press, may 2013.

[ZH02] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.

[ZJP⁺14] Daniele Zuddas, Wei Jin, Fabrizio Pastore, Leonardo Mariani, and Alessandro Orso. MIMIC: locating and understanding bugs by analyzing mimicked executions. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, pages 815–826, New York, New York, USA, sep 2014. ACM Press.

[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.